VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace funkce *If-then-else* s lazy evaluací dle specifikace v TIL

Implementation of the *If-then-else* Function with Lazy Evaluation Based on TIL

2014 Martin Velek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Martin Velek**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Implementace funkce if-then-else s lazy evaluací dle specifikace v TIL
Implementation of the If-then-else Function with Lazy Evaluation Based on TIL

Zásady pro vypracování:

Funkce "If-then-else" bývá často (zejména v Informatice) charakterizována jako "non-strict" funkce, tedy funkce, která nezachovává princip kompozicionality. V Transparentní Intensionální Logice (TIL) jsme však ukázali, že tato funkce je striktní a specifikovali jsme její rigorosní logickou definici. Na základě této definice lze pak funkci korektně implementovat.
Práce bude obsahovat:
1. Stručný přehled hlavních zásad a definic TIL.
2. TIL definice funkce "If-then-else" a její přesný výklad.
3. Převod této definice do komputační varianty TIL, což je funkcionální programovací jazyk TIL-Script.
4. Implementace funkce "If-then-else" v rámci jazyka TIL-Script.
5. Analýza otázek spojených s presupozicí a jejich zpracování v TIL-Script.
6. Příklady použití funkce, a to zejména příklady komunikace "otázka - odpověď" agentů v multi-agentním systému, kde daná otázka je spojena s presupozicí.

Seznam doporučené odborné literatury:

[1] Duží M, Jespersen B. and Materna P. (2010): Procedural Semantics for Hyperintensional Logic. Foundations and Applications of Trasnsparent Intensional Logic. First edition. Berlin: Springer, series Logic, Epistemology, and the Unity of Science, vol. 17, ISBN 978-90-481-8811-6
[2] Duží M., Materna P. (2012): TIL jako procedurální logika (průvodce zvídavého čtenáře Transparentní intensionální logikou). Aleph Bratislava 2012, ISBN 978-80-89491-08-7
[3] Duží, M., Číhalová, M., Menšík, M.: Communication in a multi-agent system; questions and answers. To appear.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. RNDr. Marie Duží, CSc.**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2014 .....

Rád bych na tomto místě poděkoval doc. RNDr. Marii Duží, CSc. za cenné podněty, rady a připomínky, které mi po celou dobu psaní této práce vždy ráda poskytla. Také děkuji Mgr. Markovi Menšíkovi, Ph.D. za přínosné konzultace. Dále děkuji své rodině, přátelům a známým, kteří mě podporovali nejen během tvorby této práce, ale po celou dobu studia.

Abstrakt

Cílem práce je podrobně popsat problematiku spjatou se specifikací funkce *If-then-else* v různých oblastech jejího využití, uvést striktní definici této funkce v Transparentní intensionální logice (TIL), ukázat její užití při analýze vět spojených s presupozicí a na příkladu komunikace agentů v multiagentním systému demonstrovat vhodnost použití TIL se striktní definicí *If-then-else* jakožto prostředku komunikace mezi agenty v multiagentním systému. Součástí práce bude rovněž shrnutí základů TIL, popis její výpočetní varianty – jazyka TIL-Script – a implementace funkce *If-then-else* (pro demonstrační účely) v rámci TIL-Scriptu.

Klíčová slova: TIL, Transparentní intensionální logika, TIL-Script, If-then-else, If-then-else-fail, lazy evaluace, MAS, multiagentní systém, komunikace v multiagentním systému.

Abstract

The aim of the thesis is to describe issues associated with the specification of the *If-then-else* function in various fields of application, introduce the strict definition of the function in Transparent Intensional Logic (TIL) and show its use at the analysis of sentences that come with a presupposition. An example of communication in a multi-agent system will demonstrate suitability of using TIL with its strict definition of *If-then-else* as means of communication of agents in multi-agent systems. The thesis will also include a brief summary of the foundations of TIL, description of its computational variant – the TIL-Script language – and implementation of the *If-then-else* function (for demonstration purposes) within TIL-Script.

Keywords: TIL, Transparent Intensional Logic, TIL-Script, If-then-else, If-then-else-fail, lazy evaluation, MAS, multi-agent system, communication in a multi-agent system.

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
ASCII	–	American Standard Code for Information Interchange
AST	–	Abstract Syntax Tree
EBNF	–	Extended Backus-Naur Form
GPS	–	Global Positioning System
IDE	–	Integrated Development Environment
TIL	–	Transparentní intensionální logika

Obsah

1 Úvod **3**
1.1 Obsah jednotlivých kapitol . . . 3

2 Stručný úvod do TIL **5**
2.1 Základní definice a pojmy . . . 5
2.1.1 Sémantické schéma . . . 5
2.1.2 Objektová báze . . . 6
2.1.3 Typy řádu 1 . . . 6
2.1.4 Intenze a extenze . . . 6
2.1.5 Empirické a analytické výrazy . . . 7
2.1.6 Konstrukce . . . 7
2.1.7 Rozvětvená hierarchie typů . . . 9
2.1.8 Nevlastní konstrukce . . . 9
2.1.9 Kvantifikátory . . . 10
2.1.10 Singularizátor . . . 11
2.2 Metoda analýzy . . . 12
2.3 Notace . . . 13

3 TIL-Script: výpočetní varianta TIL **15**
3.1 Syntax . . . 15
3.1.1 Konstrukce . . . 15
3.1.2 Typy . . . 16
3.1.3 Speciální funkce . . . 17
3.1.4 Konstanty . . . 18
3.1.5 Komentáře . . . 19
3.2 Gramatika . . . 19
3.2.1 Rozvinutá Backus-Naurova forma . . . 19
3.2.2 Vlastní gramatika v EBNF . . . 20

4 Funkce *If-then-else* **25**
4.1 *If-then-else* v predikátové logice 1. řádu . . . 25
4.1.1 Sémantika IF-THEN-ELSE . . . 26
4.1.2 Sémantika *if-then-else* . . . 26
4.2 *If-then-else* v teorii programovacích jazyků . . . 26
4.2.1 Striktnost a non-striktnost . . . 27
4.2.2 Výraz . . . 27
4.2.3 Redukční krok a redukční strategie . . . 28
4.2.4 Striktní vs. líná redukční strategie . . . 28
4.3 *If-then-else* v TIL . . . 29
4.3.1 Striktní definice . . . 29
4.3.2 Využití . . . 30
4.4 Komunikace agentů . . . 34

4.4.1 Obecné schéma posílaných zpráv 34
4.4.2 Demonstrace komunikace agentů 35

5 Jednoduchý interpret *If-then-else* **40**
5.1 Popis rozhraní a ovládání 40
5.2 Implementace 42
5.2.1 Balíček *main* 42
5.2.2 Balíček *gui* 43
5.3 Ukázky interpretace 44
5.4 Návrh řádné implementace 45
5.4.1 Třídní diagram 46

6 Závěr **49**
6.1 Dosažené cíle 49
6.2 Pohled do budoucna 49

7 Reference **50**

Přílohy **51**

A Příloha na CD **52**

1 Úvod

If-then-else je funkce, která podle pravdivosti vstupní podmínky vrátí jeden ze dvou dalších argumentů. Tato funkce bývá často označována jako non-striktní. Zjednodušeně lze toto tvrzení vysvětlit tak, že funkce vrací platný výstup i v případě, že je některý z jejích argumentů nedefinován (přesněji ten, který není na základě podmínky vybrán jako výstup). V programovacích jazycích se striktní sémantikou je takové chování problematické a podobné funkce nelze definovat na uživatelské úrovni. V TIL – Transparentní intensionální logice (dále jen TIL) je však striktní definice možná (díky hyperintenzionalitě TIL a principu lazy evaluace).

Cílem této práce je popsat problematiku specifikace funkce *If-then-else*, ukázat přednosti její specifikace v TIL, možné aplikace v oblasti logické analýzy přirozeného jazyka a také při komunikaci agentů v multiagentním systému. Dále bude funkce *If-then-else* implementována (pro demonstrační účely) dle její striktní definice v TIL v rámci jazyka TIL-Script, komputační variantě TIL.

1.1 Obsah jednotlivých kapitol

V druhé kapitole představíme TIL – Transparentní intensionální logiku. Jedná se o logický systém s procedurální sémantikou vhodný mimo jiné pro logickou analýzu přirozeného jazyka. Kapitola shrnuje základní definice a pojmy nutné pro orientaci v zápisech TILu a pochopení kapitol následujících. Dále je uvedena metoda analýzy jazykových výrazů v přirozeném jazyce. Nakonec je shrnuta notace a notační zjednodušení, kterých je často využíváno kvůli zpřehlednění a zvýšení srozumitelnosti zápisů v TILu.

Z TILu vychází funkcionální programovací jazyk TIL-Script, který přejímá většinu důležitých rysů tohoto logického systému. TIL-Script je podrobně popsán ve třetí kapitole. Nejprve je rozebrána syntax a na jednoduchých příkladech je demonstrováno, jakým způsobem se v TIL-Scriptu zapisují konstrukce TILu, jsou zde také popsány rozdíly mezi TIL a TIL-Scriptem. Po sekci s vysvětlením syntaxe následuje gramatika, na které stojí parser TIL-Scriptu, pro účely ilustrace je zde také uvedena derivace jednoduchého programu.

Čtvrtá kapitola je stěžejní částí práce. Věnuje se funkci *If-then-else(-fail)*, její specifikaci v predikátové logice 1. řádu, teorii programovacích jazyků a nakonec v TIL, kde jsou ukázány výhody její striktní definice. Je zde rozebrána celková problematika specifikace této funkce. Druhou část třetí kapitoly tvoří zajímavé aplikace této funkce, zejména tedy využití v analýze vět s presupozicí a potažmo i v komunikaci inteligentních agentů v multiagentním systému, kde se tyto věty mohou vyskytovat. V poslední části kapitoly je uveden příklad zmíněné komunikace agentů, který byl navrhnut tak, aby co nejlépe a nejsrozumitelněji demonstroval vhodnost použití TIL jakožto prostředku komunikace agentů. Celá ukázka komunikace agentů byla také převedena do TIL-Scriptu (součást elektronické přílohy).

Pátá kapitola popisuje ukázkovou implementaci funkce *If-then-else* v rámci TIL-Scriptu. Bohužel implementace interpretu TIL-Scriptu není zdaleka v takovém stavu (zatím pouze parser), aby se *If-then-else* modul mohl bez problému do TIL-Scriptu integrovat. Byl tedy

vytvořen pouze jednoduchý interpret TIL-Scriptu demonstrující chování *If-then-else* dle definice uvedené ve čtvrté kapitole. Součástí kapitoly je popis implementace a ovládání interpretu. V elektronické příloze jsou soubory s ukázkovými aplikacemi funkce *If-then-else* v TIL-Scriptu. V závěru kapitoly je nastíněn návrh možné budoucí řádné implementace reflektující některé potíže, které se vyskytly během implementace ukázkové.

V závěru načrtneme další možný vývoj jazyka TIL-Script.

2 Stručný úvod do TIL

Transparentní intensionální logika je logický systém vytvořený profesorem Pavlem Tichým (1936-1994)[1]. TIL, stejně jako jiné logiky, je prostředkem pro formalizaci přirozeného jazyka za účelem odvozování důsledků tvrzení a zkoumání vztahu mezi významem a denotátem jazykových výrazů, avšak od ostatních logik se v mnoha ohledech liší. Tichého logika je jedním z nejexpresivnějších logických systémů, umožňuje přesnou (doslovnou) analýzu jazyka a díky tomu se jejím použitím dokážeme vyhnout logickým chybám, ke kterým dochází aplikací logik se slabším vyjadřovacím aparátem (například predikátová logika 1. řádu).

TIL se snaží o nový a netradiční přístup k analýze přirozeného jazyka. Na rozdíl od tradičních systémů s množinově teoretickou sémantikou, TIL je λ-kalkul s *procedurální* sémantikou. Výraz vyjadřuje jako svůj význam proceduru, kterou je nutno vykonat, abychom dospěli k objektu, který daný výraz označuje. Tyto procedury jsou v TIL rigorosně definovány jako TIL konstrukce. Dále uvádím některé charakteristické rysy tohoto bohatého logického systému:

princip kompozicionality,
: Význam výrazu je jednoznačně dán významy jeho podvýrazů.

antikontextualismus,
: Přiřazení významu jednoznačnému výrazu zůstává za všech okolností stejné, nemění se v závislosti na kontextu, jako je tomu u některých jiných logik.

formalismus jako prostředek (ne jako cíl),
: Logika je nauka o abstraktních mimojazykových objektech, ne o jazyce samotném. Proto by měl zavedený formalismus usnadnit logickou analýzu, neměl by se však stát předmětem výzkumu.

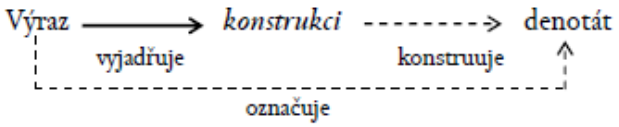
antiaktualismus.
: Aktualismus nese myšlenku, že význam výrazu je dán aktuálním světem (tj. skutečným stavem našeho světa). TIL tento názor odmítá a pracuje s pojmem tzv. možných světů. Možný svět lze pojmout jako chronologickou posloupnost množin empirických faktů, které v něm platí.

Tato kapitola vychází z [2] (pokud není uvedeno jinak). Definice, schémata a tabulky jsou převzaty beze změny, případně s drobnými úpravami, zbylý text stručně shrnuje rozsáhlejší část zmíněné knihy.

2.1 Základní definice a pojmy

2.1.1 Sémantické schéma

TIL přichází s pojmem konstrukce, abstraktní algoritmicky strukturované procedury. Každý výraz v TIL vyjadřuje konstrukci, která daný objekt (označený tímto výrazem) konstruuje (pokud nějaký takový objekt existuje). Tento objekt je denotátem daného



Obrázek 1: Sémantické schéma

výrazu. Základním typem denotátu je funkce. Lépe to ilustruje sémantické schéma 1, jež je rozšířením Fregeho sémantického trojúhelníku.

2.1.2 Objektová báze

Pro analyzovanou oblast (a tedy i jazyk, jenž danou oblast popisuje) je třeba vybrat vhodnou objektovou bázi – kolekci vzájemně disjunktních neprázdných množin. Pro přirozený jazyk se ukazuje jako nejlepší objektová báze, která obsahuje následující atomické typy:

o	množina pravdivostních hodnot $\{\mathbf{P}, \mathbf{N}\}$ (**P** – Pravda, **N** – Nepravda)
ι	množina individuí
τ	množina časových okamžiků / reálných čísel
ω	množina logicky možných světů

Tabulka 1: Objektová báze vhodná pro přirozený jazyk

2.1.3 Typy řádu 1

Nad objektovou bází můžeme budovat složitější objekty, tento proces popisuje následující induktivní definice.

Definice 2.1 Typy řádu 1

Nechť B je báze, pak:

(i) *Každý prvek B je atomický (elementární)* **typ řádu 1** *nad B.*

(ii) *Nechť* $\alpha, \beta_1, \ldots, \beta_m$ $(m > 0)$ *jsou* **typy řádu 1** *nad B. Pak kolekce* $(\alpha\beta_1 \ldots \beta_m)$ *všech m-árních parciálních funkcí, tj. zobrazení z Kartézského součinu* $\beta_1 \times \ldots \times \beta_m$ *do* α*, je molekulární (neboli funkcionální)* **typ řádu 1** *nad B.*

(iii) *Nic jiného není* **typem řádu 1** *nad B než dle (i) a (ii).*

2.1.4 Intenze a extenze

Intenze jsou objekty, které jsou závislé na možném světě, často bývají typu $((\alpha\tau)\omega)$[1] – zobrazují možné světy do chronologií objektů typu alfa $(\alpha\tau)$. Jsou označeny empirickými

[1] Takovéto typy budeme pro jednoduchost značit $\alpha_{\tau\omega}$.

výrazy jako například „Prezident ČR“. Extenze naopak na možném světě závislé nejsou. Typicky jsou označeny matematickými výrazy, například „1 + 1 = 2“.

Definice 2.2 Intenze a extenze

(α-)intenze *jsou prvky typu $(\alpha\omega)$, tedy funkce z možných světů do libovolného typu α.* **(α-)extenze** *jsou objekty typu α, kde $\alpha \neq (\beta\omega)$ pro libovolný typ β, tedy extenze jsou α-objekty, jejichž doménou není množina možných světů.*

2.1.5 Empirické a analytické výrazy

Předchozí definice intenzí a extenzí umožňuje zavést pojem empirický, resp. analytický výraz. V souvislosti s logickou analýzou přirozeného jazyka jsou tyto pojmy často používány, proto uvádím jejich definici.

Definice 2.3 Empirické a analytické výrazy, denotát a reference

Výraz, který označuje nekonstantní intenzi, tj. intenzi, která alespoň ve dvou „světamizích“ (viz. 2.3) $\langle w, t\rangle$ nabývá různých hodnot, nazveme **empirickým výrazem**. **Analytický výraz** *je výraz, jehož denotátem je buďto extenze nebo konstantní intenze nebo nemá v žádném $\langle w, t\rangle$ denotát.*

U empirických výrazů rozlišujeme **denotát** *a* **referenci v daném $\langle$w, t$\rangle$**. **Denotátem** *empirického výrazu je označená intenze I.* **Referencí** *ve $\langle w, t\rangle$ je hodnota (pokud vůbec nějaká) intenze I ve $\langle w, t\rangle$. Výraz bez denotátu může být pouze matematický výraz, a ten je vždy analytický.*

2.1.6 Konstrukce

Konstrukce můžeme chápat jako zobecnění pojmu algoritmus. Jsou to strukturované procedury, tj. skládají se z podprocedur (užitá (ne pouze zmíněná) podprocedura se také označuje termínem konstituent), a lze je provést za účelem získání nějakého výstupu. Jakožto abstraktní mimojazykové objekty jsou uchopitelné pouze skrze nějakou verbální definici. Jazykem pro zápis konstrukcí je v TIL **hyperintenzionální parciální λ-kalkul**.

Díky parcialitě pracujeme i s takovými konstrukcemi, které po provedení nedávají žádný výstup, jsou nevlastní (viz. 2.1.8). Takovéto procedury mohou být přiřazeny jako význam těm výrazům, které mají smysl, ale postrádají denotát, například „největší prvočíslo“.

Termy jazyka konstrukcí nejsou pouhou posloupností symbolů, které nabývají významu až s interpretací[2], nýbrž označují přímo konstrukce, ne tedy příslušnou funkci, která je konstruována. Z tohoto důvodu je použitý λ-kalkul hyperintenzionální[3]. TIL tedy rozlišuje mezi objekty a konstrukcemi, které tyto objekty konstruují, přičemž objektem, nad kterým konstrukce *C* operuje může být i konstrukce (ne jen objekt v pravém slova smyslu, nekonstrukce), avšak nižšího řádu než *C*. Objekty, na kterých má konstrukce operovat, jí proto musíme dodat pomocí jiné konstrukce, neboť části procedur mohou být

[2] Tak jak tomu je například v predikátové logice 1. řádu.

[3] V intenzionálních kalkulech bývá významem termů funkce konstruovaná procedurou, která daný term kóduje.

jen podprocedury. Za tímto účelem jsou zavedeny 2 atomické konstrukce, *proměnné* a *Trivializace*.

Proměnné jsou konstrukce, které konstruují objekty v závislosti na tzv. valuaci. Říkáme, že proměnné *v*-konstruují, kde *v* je parametr valuace.

Trivializace funguje jako fixní pointer na objekt. Je-li *X* objekt nějakého typu (i konstrukce), pak *Trivializace* objektu *X*, značí se ${}^0X$, konstruuje objekt *X*.

Molekulární konstrukce jsou konstrukce obsahující konstituenty jiné než sebe samé. Tyto podkonstrukce je nutné provést, pokud chceme provést danou molekulární konstrukci. Molekulární konstruce jsou tyto: *Kompozice*, *Uzávěr*, *Provedení* (*Execution*) a *Dvojí provedení* (*Double Execution*). *Kompozice* je operace aplikace funkce *f* na n-tici argumentů $A_1, \ldots, A_n$ za účelem získání hodnoty této funkce na zmíněných argumentech. Tato procedura může selhat, tj. nevyprodukovat žádný výstup, pokud daná funkce není na dodaných argumentech definována. *Uzávěr* je procedura konstruující funkci *f* abstrakcí od hodnot jejích argumentů. Provedení konstrukce *C* je ekvivalentní *C* a selže v případě, kdy *C* není konstrukcí, nebo je *C* nevlastní konstrukcí. Konstrukce vyšších řádů mohou být provedeny nadvakrát, odtud *Dvojí provedení*.

Definice 2.4 Konstrukce

(i) **Proměnná** *x je* **konstrukce**, *která konstruuje objekt O příslušného typu v závislosti na valuaci v; tedy x v-konstruuje O.*

(ii) **Trivializace**: *Je-li X jakýkoli objekt (extenze, intenze nebo i konstrukce),* ${}^0X$ *je* **konstrukce** *zvaná Trivializace. Konstruuje objekt X bez jakékoli změny.*

(iii) **Kompozice** $[X\ Y_1 \ldots Y_m]$ *je* **konstrukce**: *Je-li X konstrukce, která v-konstruuje funkci f typu* $(\alpha\beta_1 \ldots \beta_m)$, *a* $Y_1, \ldots, Y_m$ *v-konstruují po řadě objekty* $B_1, \ldots, B_m$ *typů* $\beta_1, \ldots, \beta_m$, *pak Kompozice* $[X\ Y_1 \ldots Y_m]$ *v-konstruuje hodnotu funkce f na argumentech* $B_1, \ldots, B_m$ *(tj. objekt typu* α, *pokud f má na* $\langle B_1, \ldots, B_m\rangle$ *hodnotu). Jinak je Kompozice* $[X\ Y_1 \ldots Y_m]$ *v-nevlastní, tj. ne (v-)konstruuje žádný objekt.*

(iv) **Uzávěr** $[\lambda x_1 \ldots x_m\ Y]$ *je* **konstrukce**. *Nechť* $x_1, x_2, \ldots, x_m$ *jsou navzájem různé proměnné, které v-konstruují po řadě objekty typu* $\beta_1, \ldots, \beta_m$, *a nechť Y je konstrukce, která v-konstruuje* α*-objekt. Pak* $[\lambda x_1 \ldots x_m\ Y]$ *v-konstruuje funkci f/*$(\alpha\beta_1 \ldots \beta_m)$, *a to takto: Nechť* $v(B_1/x_1, \ldots, B_m/x_m)$ *je valuace, která se liší od valuace v nanejvýš tím, že přiřazuje objekty* $B_1/\beta_1, \ldots, B_m/\beta_m$ *proměnným* $x_1, \ldots, x_m$. *Je-li Y* $v(B_1/x_1, \ldots, B_m/x_m)$*-nevlastní (viz (iii)), pak funkce f není definována na* $\langle B_1, \ldots, B_m\rangle$. *Jinak je hodnotou funkce f na argumentu* $\langle B_1, \ldots, B_m\rangle$ α*-objekt* $v(B_1/x_1, \ldots, B_m/x_m)$*-konstruovaný Y.*

(v) **Provedení** ${}^1X$ *je* **konstrukce**, *která buď v-konstruuje objekt v-konstruovaný konstrukcí X, nebo pokud X není konstrukce nebo je v-nevlastní, je rovněž* ${}^1X$ *v-nevlastní, tj. nekonstruuje žádný objekt.*

(vi) **Dvojí Provedení** ${}^2X$ *je* **konstrukce**. *Tato konstrukce je v-nevlastní, pokud X není konstrukce, nebo pokud X ne v-konstruuje jinou konstrukci, nebo v-konstruuje v-nevlastní konstrukci. Jinak, jestliže X v-konstruuje konstrukci Y a Y v-konstruuje objekt Z, pak* ${}^2X$ *v-konstruuje Z.*

(vii) *Nic jiného není* **konstrukce** *než dle (i)–(vi).*

2.1.7 Rozvětvená hierarchie typů

Abychom mohli s pomocí TIL analyzovat širší oblast přirozeného jazyka, je nutné rozšířit dříve uvedenou teorii typů 2.1. Díky tomuto rozšíření je nyní možné popsat i složitější struktury než jen „obyčejné objekty". Například ve větě „Přemysl řeší rovnici $\sin(\pi) = x$", kde objekt *Řešit* je typu $(o\iota *_n)_{\tau\omega}$ a $*_n$ je typ konstrukce (více v definici 2.5), má individuum Přemysl vztah k významu výrazu – ke konstrukci $\sin(\pi) = x$. V jednoduché teorii typů bychom tento objekt vyjádřit nedokázali.

Definice 2.5 Rozvětvená hierarchie typů nad bází B

T_1 (typy řádu 1) *byly definovány v sekci 2.1.*

C_n (konstrukce řádu n)

- **(i)** *Nechť x je proměnná, která v-konstruuje objekty typu řádu n. Pak x je* **konstrukce řádu n nad B**.
- **(ii)** *Nechť X je prvek typu řádu n. Pak* ^0X, ^1X, ^2X *jsou* **konstrukce řádu n nad B**.
- **(iii)** *Nechť* $X, X_1, \ldots, X_m$ $(m > 0)$ *jsou konstrukce řádu n nad B. Pak* $[X\ X_1 \ldots X_m]$ *je* **konstrukce řádu n nad B**.
- **(iv)** *Nechť* $x_1, \ldots, x_m, X$ $(m > 0)$ *jsou konstrukce řádu n nad B. Pak* $[\lambda x_1 \ldots x_m\ X]$ *je* **konstrukce řádu n nad B**.
- **(v)** *Nic jiného není* **konstrukce řádu n nad B** *než to, co je definováno dle* C_n *(i)–(iv).*

T_{n+1} (typy řádu n + 1) *Nechť* $*_n$ *je kolekce všech konstrukcí řádu n nad B.*

- **(i)** $*_n$ *a každý typ řádu n jsou* **typy řádu n + 1 nad B**.
- **(ii)** *Jsou-li* $\alpha, \beta_1, \ldots, \beta_m$ $(m > 0)$ *typy řádu n + 1 nad B, pak* $(\alpha\beta_1 \ldots \beta_m)$*, tj. kolekce parciálních funkcí – viz.* T_1 *(ii), je* **typ řádu n + 1** *nad B.*
- **(iii)** *Nic jiného není* **typ řádu n + 1 nad B** *než dle* T_{n+1} *(i) a (ii).*

2.1.8 Nevlastní konstrukce

Konstrukce mohou z určitých důvodů selhat. Termínem selhat je míněno prostě to, že tyto konstrukce nekonstruují žádný objekt. Takové konstrukce se nazývají (*v*-)nevlastní (anglicky „improper").

Příčinou selhání konstrukce je často pokus o aplikaci funkce na argument, na němž není daná funkce definována. Příkladem může být $[^0{:}\, x\ ^0 0]$, tato konstrukce je *v*-nevlastní pro každou valuaci proměnné x, neboť funkce dělení není definována pro žádnou dvojici čísel, kde je druhá složka (dělitel) rovna číslu 0.

Konstrukce selhávají také kvůli chybnému typování. Jestliže *C* není konstrukce řádu alespoň 1, pak je ^1C nevlastní; analogicky pro *C* řádu nižšího než 2 je ^2C nevlastní. Pokud

$X, Y_1, \ldots, Y_m$ nejsou konstrukce typově vyhovující definici 2.4 (iii), pak je i $[X\ Y_1 \ldots Y_m]$ nevlastní.

Trivializace (například ${}^0Martin$, ^{05}, ...) však není nikdy nevlastní. Stejně je tomu v případě *Uzávěru*. Může se stát, že funkce konstruovaná *Uzávěrem* nemá hodnotu pro žádný vstup, takovou funkci označíme za degenerovanou. Příkladem budiž $[\lambda x\,[{}^0\!: x\ {}^00]]$.

Protože je analýza v TIL striktně kompozicionální, uplatňuje se pravidlo „propagace parciality nahoru". Znamená to, že je-li některý z konstituentů dané *Kompozice* v-nevlastní, je v-nevlastní celá *Kompozice*.

2.1.9 Kvantifikátory

Často potřebným konstruktem logické analýzy přirozeného jazyka jsou kvantifikátory, bez nich bychom nemohli analyzovat věty typu „Všichni matematici jsou lidé". Na rozdíl od predikátové logiky jsou v TIL kvantifikátory přesně definovány jako všechny ostatní objekty, jsou to tedy funkce.

Všeobecný kvantifikátor $\forall^\alpha$ je polymorfní funkce typu $(o(o\alpha))$, kde α je libovolný typ (odtud označení polymorfní). Pro danou množinu α-prvků rozhodne, zda je úplná, tj. obsahuje-li všechny prvky typu α. *Existenční kvantifikátor* je $\exists^\alpha$ rovněž polymorfní funkce stejného typu jako *Všeobecný kvantifikátor*. Pro danou množinu α prvků zkontroluje, zda je tato množina neprázdná.

Definice 2.6 Neomezené kvantifikátory

Neomezené kvantifikátory $\forall^\alpha$, $\exists^\alpha$ *jsou typově polymorfní funkce typů* $(o(o\alpha))$ *definované takto:* **Všeobecný kvantifikátor** $\forall^\alpha$ *je funkce, která přiřazuje třídě* α*-prvků* $C/(o\alpha)$ *hodnotu* $\boldsymbol{P}$*, jestliže* C *obsahuje všechny prvky typu* α*, jinak* $\boldsymbol{N}$*.* **Existenční kvantifikátor** $\exists^\alpha$ *je funkce, která přiřazuje třídě* α*-prvků* $C/(o\alpha)$ *hodnotu* $\boldsymbol{P}$*, jestliže* C *je neprázdná, jinak* $\boldsymbol{N}$*.*

Příklad 2.1
Následující dvě konstrukce jsou ekvivalentní, ilustrují použití obou kvantifikátorů. Všimněme si, že kvantifikátory nevážou proměnné, k tomu je potřeba využít *Uzávěru*, existuje však notační zkratka, která tento fakt skrývá.

$\lambda w \lambda t\,[{}^0\forall^\iota\ \lambda x\,[[{}^0Matematik_{wt}\ x] \supset [{}^0Člověk_{wt}\ x]]]$
$\lambda w \lambda t\,[\neg[{}^0\exists^\iota\ \lambda x\,[[{}^0Matematik_{wt}\ x] \wedge \neg[{}^0Člověk_{wt}\ x]]]]$

Typy: *Člověk*, *Matematik*$/(o\iota)_{\tau\omega}$; $\supset, \wedge/(ooo)$; $\neg/(oo)$. ■

Poznámka 2.1 Předchozí příklad i příklady následující využívají zjednodušenou notaci pro lepší čitelnost. Popis notačních zkratek je v sekci 2.3.

Výše uvedené kvantifikátory však nemůžeme použít pro logickou analýzu vět v přirozeném jazyce, pokud se v dané větě nevyskytuje výraz přímo označující kvantifikátory ($\exists$, $\forall$) či spojku implikace ($\supset$), negace ($\neg$) nebo konjunkce ($\wedge$). Bylo by to v rozporu s Parmenidovým principem, který TIL přijala za vlastní (2.2). Provádíme tedy doslovnou analýzu. Analýzy z příkladu 2.2 odpovídají větám po řadě „Pro všechna individua x

platí, že je-li x matematikem, pak je i člověkem“ a „Neexistuje individuum x takové, že x je matematikem a není člověkem“. Tyto poněkud technické formulace bychom však v běžné řeči vyjádřili srozumitelněji jako „Všichni matematici jsou lidé“. Abychom mohli podobné věty analyzovat v souladu s Parmenidovým principem, zavedeme tzv. omezené kvantifikátory.

Definice 2.7 Omezené kvatifikátory

Omezené kvantifikátory All$^\alpha$ *(všichni, všechna, atd.)*, **Some$^\alpha$** *(někteří, nekterá, atd.)*, **No$^\alpha$** *(žádný, . . .)*[4] *jsou typově polymorfní funkce typů* $((o(o\alpha))(o\alpha))$, *definované takto: Je-li M/*$(o\alpha)$ *množina* α*-objektů, pak* All^α *aplikováno na M vrací množinu všech nadmnožin M,* $Some^\alpha$ *vrací množinu všech těch množin, které mají s M neprázdný průnik a* No^α *množinu těch množin, které jsou s M disjunktní.*

Nechť jsou M a N množiny stejného typu, pak:

- $[[^0All^\alpha\ M]\ N]$ *konstruuje* $\boldsymbol{P}$*, je-li* $M \subseteq N$*;*
- $[[^0Some^\alpha\ M]\ N]$ *konstruuje* $\boldsymbol{P}$*, je-li* $M \cap N \neq \emptyset$*;*
- $[[^0No^\alpha\ M]\ N]$ *konstruuje* $\boldsymbol{P}$*, je-li* $M \cap N = \emptyset$*.*

Příklad 2.2
Doslovná analýza věty „Všichni matematici jsou lidé“ je díky omezeným kvantifikátorům přímočará.

$\lambda w \lambda t\,[[^0All^\iota\ {}^0Matematik_{wt}]\ {}^0Člověk_{wt}]$

Typy: *Člověk*, *Matematik*$/(o\iota)_{\tau\omega}$; ■

2.1.10 Singularizátor

Další důležitou funkcí je *Singularizátor* I^α, v TIL modeluje výraz „ten jediný . . . “. Využívá se při analýze vět označujících nějakou individuovou roli (úřad), tj. objekt typu $\iota_{\tau\omega}$. *Singularizátor* uplatníme i později u striktní definice *If-then-else* v TIL. Je to funkce vracející jediný prvek dané množiny, obsahuje-li pouze tento prvek. Pokud množina obsahuje více prvků nebo žádný, *Singularizátor* selhává.

Definice 2.8 Singularizátor

Singularizátor I$^\alpha$ *je parciální, typově polymorfní funkce typu* $(\alpha(o\alpha))$, *která přiřazuje jednoprvkové množině C její jediný prvek, jinak (tj. pokud je C prázdná nebo obsahuje více prvků) je* I^α *nedefinován.*

[4] Názvy omezených kvantifikátorů jsou v angličtině, abychom předešli problémům se skloňováním v češtině.

Příklad 2.3
Větu „Ten jediný člověk, který dokázal Velkou Fermatovu větu“ bychom s použitím *Singularizátoru* analyzovali takto[5]:

$\lambda w \lambda t\, [{}^0I^\iota\ \lambda x\, [{}^0\mathit{Dokázat}_{wt}\ x\ {}^0\mathit{VFT}]]$.

Typy: $\mathit{Dokázat}/(o\iota\iota)_{\tau\omega}$; VFT/ι. ■

2.2 Metoda analýzy

K dosažení adekvátní analýzy je potřeba nalézt takovou konstrukci, která je daným výrazem zakódovaná a zároveň by tato konstrukce neměla obsahovat konstituenty, které výraz nezmiňuje – řídíme se tzv. Parmenidovým principem, jak jej označil sám tvůrce TIL prof. Pavel Tichý. Toto pravidlo říká, že konstituenty významu daného výrazu mohou být jen konstrukce těch objektů, které jsou ve výrazu zmíněny.

Je potřeba, aby naše analýza nebyla příliš „hrubá“. Věta „Martin cvičí“ jako celek označuje propozici $\mathit{Martin\_cvičí}/o_{\tau\omega}$. Pokud bychom této větě přidělili analýzu ${}^0\mathit{Martin\_cvičí}$, nebyli bychom schopni vyvodit z ní žádné důsledky (například, že existuje někdo, kdo cvičí), navíc by všechny ekvivalentní věty označující tutéž propozici (například „Není pravda, že Martin necvičí “) měly stejný význam – obdržely by tutéž analýzu (bez ohledu na to, jak propozici pojmenujeme, stále to bude jedna a tatáž). Abychom došli k dostatečně jemné analýze výrazu, byla zavedena tříkroková metoda analýzy:

1. *Typová analýza* objektů, o kterých daný výraz V mluví, tj. těch objektů, které jsou označeny podvýrazy výrazu V se samostatným významem.
2. *Syntéza*, tj. „poskládání“ konstrukcí objektů ad 1. tak, abychom obdrželi konstrukci objektu označeného výrazem V.
3. *Typová kontrola*, tj. kontrolujeme, zda byla syntéza provedena v souladu s typovými pravidly vyplývajícími z definice konstrukcí 2.4.

Příklad 2.4
Analyzovaná věta: Agent A přijíždí v 11.00 do ostravského servisního centra.

Typová analýza: $\mathit{Přijet}/(o\iota\iota)_{\tau\omega}$; $\mathit{Čas\_11\_00}/\tau$; A, OSC/ι.

Syntéza: $\lambda w\, [[[{}^0\mathit{Přijet}\ w]\ {}^0\mathit{Čas\_11\_00}]\ {}^0A\ {}^0\mathit{OSC}]$.

Přijet je intenze, kterou je třeba nejprve extenzionalizovat aplikací aktuálního světa w a času, za který však dosadíme časový okamžik, kdy individum A přijíždí do servisního centra, tj. *Čas_11_00*. Zbývá určit, kdo má přijet kam, tedy postupně A a OSC.

[5] Konstrukce konstruuje individuový úřad, který v našem světočase zastává Andrew Wiles.

Typová kontrola:

(a) ${}^0\textit{Přijet} \to (o\iota\iota)_{\tau\omega}$,

(b) $[{}^0\textit{Přijet}\ w] \to_v (o\iota\iota)_\tau$,

(c) $[[{}^0\textit{Přijet}\ w]\ {}^0\textit{Čas\_11\_00}] \to_v (o\iota\iota)$,

(d) $[[[{}^0\textit{Přijet}\ w]\ {}^0\textit{Čas\_11\_00}]\ {}^0A] \to_v (o\iota)$,

(e) $[[[{}^0\textit{Přijet}\ w]\ {}^0\textit{Čas\_11\_00}]\ {}^0A\ {}^0\textit{OSC}] \to_v o$,

(f) $\lambda w\,[[[{}^0\textit{Přijet}\ w]\ {}^0\textit{Čas\_11\_00}]\ {}^0A\ {}^0\textit{OSC}] \to o_\omega$.

Intenze *Přijet* konstruuje **P** nebo **N** v závislosti na světě a na tom jestli dané individuum *A* přijíždí do ostravského servisního centra (zde také typu ι) v daném čase. Celá věta tedy vyjadřuje konstrukci typu o_ω. Postupnou aplikací argumentů bychom tedy měli k tomuto typu dojít. V posledním kroku abstrahujeme od hodnot proměnné w, neboť empirické výrazy (analyzovaná věta) označují intenze, které mohou být obecně nekonstantní, a ne jejich náhodné hodnoty v daném stavu světa.

■

2.3 Notace

Vzhledem ke složitosti zápisu konstrukcí v TIL, je na místě zmínit pár zjednodušujících zásad, které tyto zápisy pomohou zpřehlednit.

- Vnější závorky [...] lze vynechat tam, kde jejich odstranění nepovede k nejednoznačné interpretaci.
- Jakožto proměnné v-konstruující prvky typu ω (možné světy) je vhodné použít $w, w_1, w_2, \ldots, w_n$. Obdobně to platí pro prvky typu τ (čísla / časové okamžiky) a proměnné $t, t_1, t_2, \ldots, t_n$.
- *Kompozice* tvaru $[[C\,w]\,t]$, kde C v-konstruuje α-intenzi je možno zkrátit na C_{wt}, jedná se o tzv. extenzionální sestup nebo extenzionalizaci intenze.
- Dvojici možný svět a čas nazveme „světočas“ či „světamih“ a budeme používat značení $\langle w, t\rangle$.
- TIL se drží prefixového zápisu, avšak zejména u složitějších konstrukcí takový zápis snižuje čitelnost, proto se zavedené operátory ($+$, $-$, $=$, $\wedge$, $\vee$, $\supset$, $\neg$[6], ...) používají infixně a bez *Trivializace*. Například tuto *Kompozici* $[{}^0\vee\ [{}^0\wedge\ p\ q]\ [[{}^0\neg\ q]]]$ lze zapsat jednodušeji takto $[[p \wedge q] \vee \neg q]$.

[6] Symbol negace lze zapsat bezprostředně před negovaný objekt.

- Neomezené kvantifikátory a jejich proměnné[7] lze také zapisovat stylem známým z predikátové logiky, tj. místo $[{}^0\forall^\alpha\ \lambda x\ M]$, $[{}^0\exists^\alpha\ \lambda x\ M]$ můžeme psát $[\forall x\ M]$ a $[\exists x\ M]$.
- Aplikaci *Singularizátoru* můžeme zpřehlednit zápisem $[\iota x\ M]$ oproti původnímu $[{}^0I^\iota\ \lambda x\ M]$.

[7] Kvantifikátory v TIL samozřejmě proměnné neváží, asociace proměnných s kvantifikátory je pouze záležitost notace a slouží k lepší čitelnosti zápisu.

3 TIL-Script: výpočetní varianta TIL

TIL-Script je nekonvenční funkcionální programovací jazyk založený na TIL. Cílem projektu bylo možné využití při analýze přirozeného jazyka. Další uplatnění TIL-Script nalezl jakožto prostředek komunikace v multiagentním systému. TIL nebyl navržen jako programovací jazyk a některá jeho specifika (například používání symbolů nepatřících do ASCII, chybějící základní datové struktury – celá čísla, řetězce, seznamy, . . .) musela být pozměněna, v důsledku toho se TIL-Script poněkud liší od původního TIL.

Následující odstavce se postupně věnují popisu základní syntaxe TIL-Scriptu a upozorňují na zmíněné rozdíly tak, jak je uvedeno v [4]. Je rozebrána také gramatika TIL-Scriptu, po které následuje shrnutí odlišností od původní TIL.

3.1 Syntax

3.1.1 Konstrukce

Proměnné Proměnné jsou posloupnosti znaků začínající malým písmenem, mohou obsahovat pouze alfanumerické znaky a „_“. Za prvním výskytem každé proměnné musí být (po dvojtečce „:“) uveden její typ, výjimkou jsou proměnné deklarované globálně (před samotným použitím).

Trivializace Trivializace se označuje symbolem apostrofu „’“. Od Trivializace v TIL se jinak nijak neliší, jakákoliv konstrukce může být trivializována.

Uzávěr Uzávěr používá „\“ namísto řeckého λ (lambda), avšak za názvem proměnné musí po dvojtečce „:“ následovat její typ. Často užívanými proměnnými jsou v TIL w a t, které značí možný svět a čas, z tohoto důvodu jsou v TIL-Scriptu tyto názvy proměnných vyhrazeny pro typy World a Time (více v sekci 3.1.2) a nemusí se tedy za těmito proměnnými uvádět.

Příklad 3.1
Funkce následníka v TIL-Scriptu.

```
[\x:Int ['+ x '1]].
```

■

Pojmenované funkce V TIL-Scriptu lze definovat i pojmenované funkce. Slouží k tomu klíčové slovo **Def** následované názvem funkce a dvojicí znaků „:=“ pro přiřazení *Uzávěru*, který funkci definuje. Pojmenování se vztahuje ke konstrukci, termín pojmenovaná funkce je tedy mírně zavádějící. Použití pojmenované funkce se obejde již bez *Trivializace* – jméno funkce se nahradí příslušnou konstrukcí (kdyby to byla přímo ona funkce, musela by být nejprve trivializována).

Příklad 3.2
Funkce následníka Succ v TIL-Scriptu a její volání.

```
Def Succ := [\x:Int ['+ x '1]].
[Succ '1].
```

■

Kompozice *Kompozice* se v ničem neliší, avšak nelze použít zjednodušení syntaxe, jak je uvedeno v 2.3, a je potřeba důsledně používat závorky [...].

Dvojí provedení *Dvojí provedení* není v TIL-Scriptu realizováno přímo, místo toho je zaveden koncept n-tého *Provedení*. Konstrukce se provede, poté se provede konstrukce, která je výsledkem přecházejícího *Provedení*, to se opakuje n-krát. Pokud některé z *Provedení* (kromě posledního) nedá na výstup konstrukci, celé n-té *Provedení* je *v*-nevlastní. *Provedení* konstrukce *C* n-krát se značí `^nC`, *Dvojí provedení* se pak zapíše jako `^2C`.

3.1.2 Typy

Často používaná objektová báze (o, ι, τ, ω) je v TIL-Scriptu zavedena [8] a rozšířena o několik dalších typů běžně používaných v programování. Často používané typy intenzí `(Any Time World)` (v TIL $\alpha_{\tau\omega}$) lze zapsat takto: `(Any)@tw`. Pro extenzionalizaci intenzí lze využít notační zkratky `@wt`. Srovnání typů v TIL a TIL-Scriptu zachycuje tabulka 2.

Typ v TIL	Typ v TIL-Scriptu	Popis
o	Bool	množina pravdivostních hodnot True a False
ι	Indiv	množina individuí
τ	Time	množina časových okamžiků
ω	World	množina možných světů
—	Int	množina celých čísel
—	Real	množina reálných čísel
—	String	typ textového řetězce
α	Any	nespecifikovaný typ (libovolný typ alfa)
$*_n$	$*$	typ konstrukcí

Tabulka 2: Přehled typů v TIL a TIL-Scriptu

[8] Místo obtížně použitelných řeckých písmen jsou použita klíčová slova.

Seznamy a n-tice Seznam (potenciálně nekonečná n-tice) je datový typ hojně využívaný v programování. Je to jedna ze základních datových struktur, nepřekvapí proto, že se jej autoři rozhodli přidat i do TIL-Scriptu. V TIL takovéto struktury chybí, protože je možno je definovat jakožto funkce. Nicméně, z důvodu jejich častého používání je výhodnější mít možnost zadat přímo typ seznamu. Seznam se deklaruje použitím klíčového slova **List** a v kulatých závorkách se uvede typ jeho prvků, například `List(Indiv)` je seznam individuí.

N-tice je možno deklarovat s pomocí klíčového slova **Tuple** a v závorkách uvedených typů oddělených čárkou, například `Tuple(Int, Bool)` je uspořádaná dvojice celé číslo – pravdivostní hodnota.

Deklarace typů objektů Objekty musí mít v TIL přiděleny svůj typ, stejné je to pochopitelně i v případě TIL-Scriptu. Deklarace typu objektu začíná jeho názvem, následuje znak „/“ a za ním příslušný typ. Objektům (odděleným čárkou) lze přidělit typ najednou. Proměnné jsou typu $*_n$, v jejich případě nás zajímá zejména co konstruují (popsáno v 3.1.1).

Příklad 3.3
Deklarace typu funkce PresidentOf v TIL-Scriptu.

```
PresidentOf/(Indiv Indiv)@tw.
```

■

Pojmenované typy Stejně jako můžeme z *Uzávěrů* tvořit pojmenované funkce s použitím klíčového slova **Def**, lze vytvořit i pojmenované typy a to díky klíčovému slovu **TypeDef**. Takto definované typy lze použít při deklaraci objektů v-konstruovaných proměnnými a jiných objektů výše popsaným způsobem (popořadě s pomocí „:“ a „/“).

Příklad 3.4
Deklarace pojmenovaného typu IndivVlast a jeho použití:

```
TypeDef IndivVlast := (Bool Indiv)@tw.
x:IndivVlast.
Student/IndivVlast.
```

■

3.1.3 Speciální funkce

Booleovské operátory Pro vybrané booleovské operace existují v TIL-Scriptu funkce dostupné pod klíčovými slovy:

- **And** (konjunkce)
- **Or** (disjunkce)

- **Not** (negace)
- **Implies** (implikace)
- **Equals** (rovnost pravdivostních hodnot)

Operátor přiřazení Operátor přiřazení spojí danou proměnnou s hodnotou pomocí klíčového slova Let a dvojice znaků „:=“.

Příklad 3.5
Přiřazení hodnoty 5 proměnné x v TIL-Scriptu.

```
Let x := '5.
```

■

Kvantifikátory TIL-Script zná oba typy (neomezených) kvantifikátorů, všeobecný i existenční. *Všeobecný kvantifikátor* se používá pomocí klíčového slova **ForAll** spolu s *Trivializací*, *Existenční kvantifikátor* se používá stejným způsobem, jen s tím rozdílem, že klíčovým slovem je v tomto případě **Exist**.

Příklad 3.6
Použití *Existenčního kvantifikátoru* v TIL-Scriptu.

```
['Exist [\x:Int ['< x '0]]].
```

■

Singularizátor *Singularizátor* je v TIL-Scriptu také zabudován. Podobně jako kvantifikátory, *Singularizátor* se používá pomocí klíčového slova **Single** s *Trivializací*. Použití je tedy stejné jako u kvantifikátorů.

Funkce pro selhání V některých případech vyžadujeme, aby celá konstrukce selhala (například při použití funkce *If-then-else-fail* – pokud není splněna podmínka, celá konstrukce musí selhat). Pro tyto případy existuje v TIL-Scriptu vestavěná funkce pod klíčovým slovem **Fail**.

3.1.4 Konstanty

V TIL-Scriptu je možné použít 3 typy konstant – může jí být číslo, řetězec nebo pravdivostní hodnota. Číselnou konstantou může být buď celé nebo desetinné číslo (float). Řetězec je posloupnost znaků uzavřená v uvozovkách (””). Pro pravdivostní hodnoty jsou určena klíčová slova **True** a **False**.

3.1.5 Komentáře

Pro účely popisu kódu jsou v TIL-Scriptu k dispozici dva druhy komentářů. Pro komentáře nepřesahující na další řádek je určen znak „#“, za nímž je vše až do konce řádku ignorováno. Víceřádkové komentáře se realizují počáteční „/*“ a koncovou „*/“ značkou, mezi tyto značky se vkládá samotný text.

3.2 Gramatika

Syntax TIL-Scriptu je dána bezkontextovou gramatikou, jež byla převzata z diplomové práce N. Cipricha [4], který se zabýval implementací parseru pro TIL-Script. Tato gramatika byla svým autorem upravena a převedena do EBNF (Extended Backus–Naur Form, česky Rozvinutá Backus-Naurova forma).

Poznámka 3.1 Současná verze TIL-Scriptu se od té původní verze v některých aspektech liší. Zde popisovaná syntax odráží nynější stav TIL-Scriptu (tak, jak se požadavky a potřeby vyvíjely s časem), avšak tato verze ještě není připravena k nasazení.

3.2.1 Rozvinutá Backus-Naurova forma

EBNF má určitá specifika týkající se zápisu pravidel gramatiky , pro lepší porozumění je proto vhodné zde tato specifika vysvětlit [5], navzdory tomu, že je nerozvinutá varianta Backus-Naurovy formy poměrně často používaná v praxi.

- Terminály a terminální řetězce jsou uzavřeny do jednoduchých uvozovek.
- Symbol „=“ má stejný význam jako šipka („→“) v klasické notaci.
- Znak svislé čáry („|“) slouží stejnému účelu jako v klasickém zápisu (odděluje alternativní přepisy daného neterminálu).
- Každé pravidlo (soubor pravidel oddělených svislou čarou) je ukončeno středníkem.
- Pro zřetězení se v EBNF používá čárka.
- Libovolné opakování řetězce terminálních a neterminálních symbolů se značí uzavřením tohoto řetězce do složených závorek („{“, „}“).
- Možnost jednoho nebo žádného výskytu řetězce terminálních a neterminálních symbolů je realizováno uzavřením tohoto řetězce do hranatých závorek („[“, „]“).
- Kulaté závorky určují prioritu derivace.
- Výrazy se zvláštním významem se vkládají mezi symboly otazníku, interpretace těchto výrazů je mimo možnosti gramatiky[9].

[9] V gramatice TIL-Scriptu je této možnosti využito při pojmenování bílých znaků – mezery, tabulátoru a znaku nového řádku – kde je interpretace problematická, ať už z notačního hlediska nebo jiného (například znak nového řádku na platformě Windows se liší od znaku nového řádku na Linuxu).

Příklad 3.7
Zápis $x = y, (\text{'}a\text{'} \mid \text{'}b\text{'})$; je ekvivalentní tomuto zápisu v klasické notaci: $X \to Ya \mid Yb$ ∎

3.2.2 Vlastní gramatika v EBNF

Nyní již můžeme uvést avizovanou gramatiku TIL-Scriptu. Značení neterminálů je kvůli přehlednosti převzato z nerozvinuté Backus-Naurovy formy, tj. každý výraz označující neterminál začíná a končí špičatou závorkou („⟨“, „⟩“)[10].

$TS\_Grammar = (\Pi, \Sigma, P, S)$

$\Pi = \{\langle start\rangle, \langle termination\rangle, \langle type\text{-}definition\rangle, \langle named\text{-}closure\rangle, \langle entity\text{-}definition\rangle,$
$\langle construction\rangle, \langle global\text{-}variable\text{-}definition\rangle, \langle data\text{-}type\rangle, \langle trivialisation\text{-}construction\rangle,$
$\langle variable\text{-}construction\rangle, \langle composition\text{-}construction\rangle, \langle closure\text{-}construction\rangle,$
$\langle lambda\text{-}variables\rangle, \langle typed\text{-}variables\rangle, \langle n\text{-}execution\text{-}construction\rangle, \langle entity\rangle, \langle type\text{-}name\rangle,$
$\langle closure\text{-}name\rangle, \langle entity\text{-}name\rangle, \langle variable\text{-}name\rangle, \langle keyword\rangle, \langle lowercase\text{-}letter\rangle,$
$\langle uppercase\text{-}letter\rangle, \langle symbols\rangle, \langle digit\rangle, \langle non\text{-}zero\text{-}digit\rangle, \langle number\rangle, \langle lowerletter\text{-}name\rangle,$
$\langle upperletter\text{-}name\rangle, \langle whitespace\rangle, \langle optional\text{-}whitespace\rangle, \langle whitespace\text{-}character\rangle\}$

$\Sigma = \{\text{'}a\text{'}, \text{'}b\text{'}, \ldots, \text{'}z\text{'}, \text{'}A\text{'}, \text{'}B\text{'}, \ldots, \text{'}Z\text{'}, \text{'}0\text{'}, \text{'}1\text{'}, \ldots, \text{'}9\text{'}, \text{'}.\text{'}, \text{'},\text{'}, \text{':'}, \text{'}=\text{'}, \text{'}+\text{'}, \text{'}-\text{'}, \text{'}\_\text{'}, \text{'}/\text{'},$
$\text{'}\backslash\text{'}, \text{'}*\text{'}, \text{'}@\text{'}, \text{'}(\text{'}, \text{')'}, \text{'}[\text{'}, \text{']'}, \text{'}\hat{}\text{'}, SPACE, TAB, NEWLINE\}$

$P = \{$

⟨*start*⟩ = { ⟨*type-definition*⟩, ⟨*termination*⟩
| ⟨*entity-definition*⟩, ⟨*termination*⟩
| ⟨*construction*⟩, ⟨*termination*⟩
| ⟨*named-closure*⟩, ⟨*termination*⟩
| ⟨*global-variable-definition*⟩, ⟨*termination*⟩ };

⟨*termination*⟩ = ⟨*optional-whitespace*⟩, '`.`';

⟨*type-definition*⟩ = '`TypeDef`', ⟨*whitespace*⟩, ⟨*type-name*⟩,
⟨*optional-whitespace*⟩, '`:=`', ⟨*optional-whitespace*⟩, ⟨*data-type*⟩;

⟨*named-closure*⟩ = '`Def`', ⟨*whitespace*⟩, ⟨*closure-name*⟩,
⟨*optional-whitespace*⟩, '`:=`', ⟨*optional-whitespace*⟩, ⟨*closure-construction*⟩;

⟨*entity-definition*⟩ = ⟨*entity-name*⟩, ⟨*optional-whitespace*⟩,
{ '`,`', ⟨*optional-whitespace*⟩, ⟨*entity-name*⟩ }, '`/`',
⟨*optional-whitespace*⟩, (⟨*data-type*⟩ | ⟨*type-name*⟩);

⟨*construction*⟩ = ⟨*trivialisation-construction*⟩
| ⟨*variable-construction*⟩

[10] V EBNF však tyto závorky nejsou nutné.

| ⟨*composition-construction*⟩
| ⟨*closure-construction*⟩
| ⟨*n-execution-construction*⟩
| ⟨*entity*⟩;

⟨*global-variable-definition*⟩ = ⟨*variable-name*⟩, { ⟨*optional-whitespace*⟩, ',', ⟨*optional-whitespace*⟩, ⟨*variable-name*⟩ }, ⟨*optional-whitespace*⟩, ':', ⟨*optional-whitespace*⟩, ⟨*data-type*⟩;

⟨*data-type*⟩ = 'Bool'
| 'Indiv'
| 'Time'
| 'String'
| 'World'
| 'Real'
| 'Int'
| 'Any'
| '*'
| 'List', ⟨*optional-whitespace*⟩, '(', ⟨*optional-whitespace*⟩, ⟨*data-type*⟩, ⟨*optional-whitespace*⟩, ')'
| 'Tuple', ⟨*optional-whitespace*⟩, '(', ⟨*optional-whitespace*⟩, ⟨*data-type*⟩, ⟨*optional-whitespace*⟩, ')'
| '(', ⟨*optional-whitespace*⟩, ⟨*data-type*⟩, ⟨*optional-whitespace*⟩ ')'
| ⟨*data-type*⟩, ⟨*data-type*⟩
| ⟨*data-type*⟩, '@tw';

⟨*trivialisation-construction*⟩ = '', ⟨*construction*⟩, ['@wt'];

⟨*variable-construction*⟩ = ⟨*variable-name*⟩;

⟨*composition-construction*⟩ = '[', ⟨*optional-whitespace*⟩, ⟨*construction*⟩, ⟨*construction*⟩, { ⟨*construction*⟩ }, ⟨*optional-whitespace*⟩, ']';

⟨*closure-construction*⟩ = '[', ⟨*optional-whitespace*⟩, ⟨*lambda-variables*⟩, ⟨*optional-whitespace*⟩, ⟨*construction*⟩, ']';

⟨*lambda-variables*⟩ = '\', ⟨*typed-variables*⟩;

⟨*typed-variables*⟩ = ⟨*variable-name*⟩, ⟨*optional-whitespace*⟩, { ',', ⟨*variable-name*⟩ }, [⟨*optional-whitespace*⟩, ':', ⟨*optional-whitespace*⟩, ⟨*data-type*⟩];

⟨*n-execution-construction*⟩ = '^', ⟨*non-zero-digit*⟩, ⟨*optional-whitespace*⟩, ⟨*construction*⟩;

⟨*entity*⟩ = ⟨*keyword*⟩
| ⟨*entity-name*⟩

| ⟨*number*⟩
| ⟨*symbol*⟩;

⟨*type-name*⟩ = ⟨*upperletter-name*⟩;

⟨*closure-name*⟩ = ⟨*upperletter-name*⟩;

⟨*entity-name*⟩ = ⟨*upperletter-name*⟩;

⟨*variable-name*⟩ = ⟨*lowerletter-name*⟩;

⟨*keyword*⟩ = 'ForAll'
| 'Exist'
| 'Single'
| 'Every'
| 'Some'
| 'True'
| 'False'
| 'And'
| 'Or'
| 'Not'
| 'Implies'
| 'Equals'
| 'Fail';

⟨*lowercase-letter*⟩ = 'a'
| 'b'
⋮
| 'z';

⟨*uppercase-letter*⟩ = 'A'
| 'B'
⋮
| 'Z';

⟨*symbols*⟩ = '+'
| '-'
| '*'
| '/';

⟨*digit*⟩ = '0'
| ⟨*non-zero-digit*⟩;

⟨*non-zero-digit*⟩ = '1'
| '2'

⋮
| ‘9’;

⟨*number*⟩ = ⟨*non-zero-digit*⟩, { ⟨*digit*⟩ }, [‘.’, ⟨*digit*⟩, { ⟨*digit*⟩ }];

⟨*lowerletter-name*⟩ = ⟨*lowercase-letter*⟩, { ⟨*lowercase-letter*⟩ | ‘_’ | ⟨*digit*⟩ };

⟨*upperletter-name*⟩ = ⟨*uppercase-letter*⟩, { ⟨*lowercase-letter*⟩ | ‘_’ | ⟨*digit*⟩ };

⟨*whitespace*⟩ = ⟨*whitespace-character*⟩, ⟨*optional-whitespace*⟩;

⟨*optional-whitespace*⟩ = { ⟨*whitespace-character*⟩ };

⟨*whitespace-character*⟩ = *? SPACE ?*
| *? TAB ?*
| *? NEWLINE ?*;

}

$S = \langle start \rangle$

Ukázka derivace Na jednoduchém příkladě nyní ilustrujme schopnost gramatiky generovat syntakticky validní programy v TIL-Scriptu. Generovaným programem bude definice typu propozice: `TypeDef Prop := Bool@tw.`

⟨*start*⟩ ⇒
⟨*type-definition*⟩, ⟨*termination*⟩ ⇒
‘`TypeDef`’, ⟨*whitespace*⟩, ⟨*type-name*⟩, ⟨*optional-whitespace*⟩, ‘**:=**’, ⟨*optional-whitespace*⟩,
⟨*data-type*⟩, ⟨*termination*⟩ ⇒
‘`TypeDef`’, ⟨*whitespace-character*⟩, ⟨*optional-whitespace*⟩, ⟨*type-name*⟩, ⟨*optional-whitespace*⟩,
‘**:=**’, ⟨*optional-whitespace*⟩, ⟨*data-type*⟩, ⟨*termination*⟩ ⇒
‘`TypeDef`’, *? SPACE ?*, ⟨*optional-whitespace*⟩, ⟨*type-name*⟩, ⟨*optional-whitespace*⟩, ‘**:=**’,
⟨*optional-whitespace*⟩, ⟨*data-type*⟩, ⟨*termination*⟩ ⇒
‘`TypeDef`’, *? SPACE ?*, ⟨*type-name*⟩, ⟨*optional-whitespace*⟩, ‘**:=**’, ⟨*optional-whitespace*⟩,
⟨*data-type*⟩, ⟨*termination*⟩ ⇒
‘`TypeDef`’, *? SPACE ?*, ⟨*upperletter-name*⟩, ⟨*optional-whitespace*⟩, ‘**:=**’, ⟨*optional-whitespace*⟩,
⟨*data-type*⟩, ⟨*termination*⟩ ⇒
‘`TypeDef`’, *? SPACE ?*, ⟨*uppercase-letter*⟩, ⟨*lowercase-letter*⟩, ⟨*lowercase-letter*⟩,
⟨*lowercase-letter*⟩, ⟨*optional-whitespace*⟩, ‘**:=**’, ⟨*optional-whitespace*⟩, ⟨*data-type*⟩,
⟨*termination*⟩ ⇒

‘TypeDef’, *? SPACE ?*, ‘P’, ⟨*lowercase-letter*⟩, ⟨*lowercase-letter*⟩, ⟨*lowercase-letter*⟩,
⟨*optional-whitespace*⟩, ‘:=’, ⟨*optional-whitespace*⟩, ⟨*data-type*⟩, ⟨*termination*⟩ ⇒

‘TypeDef’, *? SPACE ?*, ‘P’, ‘r’, ⟨*lowercase-letter*⟩, ⟨*lowercase-letter*⟩, ⟨*optional-whitespace*⟩,
‘:=’, ⟨*optional-whitespace*⟩, ⟨*data-type*⟩, ⟨*termination*⟩ ⇒

‘TypeDef’, *? SPACE ?*, ‘P’, ‘r’, ‘o’, ⟨*lowercase-letter*⟩, ⟨*optional-whitespace*⟩, ‘:=’,
⟨*optional-whitespace*⟩, ⟨*data-type*⟩, ⟨*termination*⟩ ⇒

‘TypeDef’, *? SPACE ?*, ‘P’, ‘r’, ‘o’, ‘p’, ⟨*optional-whitespace*⟩, ‘:=’, ⟨*optional-whitespace*⟩,
⟨*data-type*⟩, ⟨*termination*⟩ ⇒

‘TypeDef’, *? SPACE ?*, ‘P’, ‘r’, ‘o’, ‘p’, *? SPACE ?*, ‘:=’, ⟨*optional-whitespace*⟩, ⟨*data-type*⟩,
⟨*termination*⟩ ⇒

‘TypeDef’, *? SPACE ?*, ‘P’, ‘r’, ‘o’, ‘p’, *? SPACE ?*, ‘:=’, *? SPACE ?*, ⟨*data-type*⟩, ‘@tw’
⟨*termination*⟩ ⇒

‘TypeDef’, *? SPACE ?*, ‘P’, ‘r’, ‘o’, ‘p’, *? SPACE ?*, ‘:=’, *? SPACE ?*, ‘Bool’, ‘@tw’,
⟨*termination*⟩ ⇒

‘TypeDef’, *? SPACE ?*, ‘P’, ‘r’, ‘o’, ‘p’, *? SPACE ?*, ‘:=’, *? SPACE ?*, ‘Bool’, ‘@tw’,
⟨*optional-whitespace*⟩, ‘.’ ⇒

‘TypeDef’, *? SPACE ?*, ‘P’, ‘r’, ‘o’, ‘p’, *? SPACE ?*, ‘:=’, *? SPACE ?*, ‘Bool’, ‘@tw’, ‘.’

Po interpretaci speciálních znaků *SPACE* již dostaneme terminální řetězec[11] – program v TIL-Scriptu (v tomto případě pouze definici pojmenovaného typu propozice, v TIL $o_{\tau\omega}$).

[11] Pro připomenutí: čárka značí zřetězení a jednoduché uvozovky nejsou součástí terminálních řetězců.

4 Funkce *If-then-else*

If-then-else je z matematického pohledu funkce 3 argumentů, která na základě podmínky, 1. argumentu, vybere jeden ze dvou argumentů zbývajících jakožto výstup. Pokud je podmínka pravdivá, vybere se 2. argument, v opačném případě je výsledkem 3. argument. Formálně je *If-then-else* zobrazení $\{P, N\} \times M \times M \mapsto M$, kde $\{P, N\}$ je množina pravdivostních hodnot a M je libovolná množina (je to tedy polymorfní funkce). Aplikaci *If-then-else* na argumenty, matematicky *If-then-else(C, A, B)*, zapisujeme pro lepší čitelnost jako If C then A else B ($C \in \{P, N\}$, $A, B \in M$). Tato definice funkce *If-then-else* je zde pouze pro ilustraci principu jejího fungování, v praxi (například v logických systémech či v programování) je její specifikace složitější, jak uvidíme dále v kapitole.

Příklad 4.1
Zvolme množinu $M = \{1, 2\}$, jejíž prvky jsou přípustnými hodnotami 2. a 3. argumentu funkce *If-then-else*, pak je takto funkce určena předpisem:

C	P	P	P	P	N	N	N	N
A	1	1	2	2	1	1	2	2
B	1	2	1	2	1	2	1	2
$f(C, A, B)$	1	1	2	2	1	2	1	2

Tabulka 3: Funkce *If-then-else* zadaná tabulkou; $f(C, A, B)$ = If C then A else B.

■

Tato kapitola si klade za cíl popsat možné problémy se specifikací *If-then-else* a poukázat na řešení, které nabízí TIL a které bude zahrnuto v její komputační variantě TIL-Script. Následující podkapitoly se věnují problematice specifikace této funkce v predikátové logice, teorii programovacích jazyků a nakonec v TIL.

4.1 *If-then-else* v predikátové logice 1. řádu

Funkce *If-then-else* je definována v rozšířené verzi predikátové logiky 1. řádu, kde má dvojí použití [6]:

1. **logická spojka** – IF ⟨formule⟩ THEN ⟨formule⟩ ELSE ⟨formule⟩;
2. **logický operátor** – *if* ⟨formule⟩ *then* ⟨term⟩ *else* ⟨term⟩.

Poznámka 4.1 Pro označení logické spojky použijeme název IF-THEN-ELSE, zatímco logický operátor budeme mít na mysli v případě použití spojení *if-then-else*.

Rozdíl mezi spojkou a operátorem spočívá v typech argumentů této funkce. Zatímco všechny argumenty IF-THEN-ELSE jsou formule a vrací formuli, 1. parametr *if-then-else* je formule a zbylé dva jsou termy, výstupem aplikace *if-then-else* je tedy term.

4.1.1 Sémantika IF-THEN-ELSE

Význam spojky IF-THEN-ELSE je dán ternární funkcí, která zobrazuje množinu $\{P, N\} \times \{P, N\} \times \{P, N\}$ do množiny $\{P, N\}$. Tato funkce je určena vztahy:

- (IF P THEN α ELSE β) je α;
- (IF N THEN α ELSE β) je β;

kde α i β nabývají hodnot **P**, **N**.

4.1.2 Sémantika *if-then-else*

Význam operátoru *if-then-else* je dán ternární funkcí zobrazující množinu $\{P, N\} \times D \times D$ do množiny D, jež je dána následujícími vztahy. Pro $d_1, d_2 \in D$:

- (*if* P *then* d_1 *else* d_2) je d_1;
- (*if* N *then* d_1 *else* d_2) je d_2.

Logická, tj. deklarativní specifikace tohoto konstruktu v predikátové logice 1. řádu však není možná v tom případě, kdy potřebujeme zachytit i takové situace, ve kterých může provedení daného příkazu selhat. V takovém případě potřebujeme hyperintenzionální logiku parciálních funkcí, jakou je například i TIL, jejíž principy byly vysvětleny v kapitole 2. Specifikaci *If-then-else* v TIL se věnuje sekce 4.3

4.2 *If-then-else* v teorii programovacích jazyků

Zřejmě nejdůležitější uplatnění funkce *If-then-else* nalezla v programování, kde slouží jako konstrukt umožňující větvení programu[12].

Příčinou problematické specifikace sémantiky této funkce v teorii programovacích jazyků je skutečnost, že požadujeme, aby funkce v určitých případech vrátila výsledek navzdory tomu, že některý z jejích argumentů není definován. Takovéto nedefinované argumenty jsou *výrazy* (pojem *výraz* je vysvětlen v sekci 4.2.2), které budeme značit symbolem $\perp$. *Výraz* (v Haskellu) `1 `div` 0` může sloužit jako příklad takového nedefinovaného *výrazu*.

Nyní můžeme naše požadavky na výstupy funkce *If-then-else* zpřesnit. Požadujeme, aby funkce vrátila hodnotu 2. argumentu (výsledek jeho provedení) i v případě, že 3. argumentem je $\perp$ za předpokladu, že podmínka je pravdivá. Analogicky, pokud bude podmínka nepravdivá, vráceným výsledkem bude hodnota 3. argumentu, i když bude 2. argument $\perp$. Pokud je 2. i 3 argument $\perp$ nebo je-li 1. argument $\perp$, funkce selhává. Lépe to vystihuje následující zápis:

[12] V imperativních jazycích, jako je například Java, je *If-then-else* míněn jako složený příkaz (s nepovinnou else-částí), kde se po vyhodnocení podmínky vybere blok příkazů, kterými má program pokračovat. Toto pojetí funkce *If-then-else* nebude předmětem našeho zkoumání, i když jeho podstata je stejná. Zde nám však jde o logickou, deklarativní specifikaci programu.

- If **P** then A else $\perp$ = A
- If **N** then $\perp$ else A = A
- jinak $\perp$[13]

Říkáme, že funkce *If-then-else* je striktní (strict) na 1. a souběžně striktní (jointly strict) na 2. a 3. parametru. Není striktní na 2. ani na 3. parametru (na těchto parametrech je non-striktní) [7].

Nyní je potřeba definovat několik důležitých pojmů, poté z výše uvedených požadavků kladených na chování funkce *If-then-else* vyvodíme s pomocí těchto pojmů patřičné důsledky.

Poznámka 4.2 Předpokládá se funkcionální paradigma (v následujících příkladech jednoduchých programů je využit programovací jazyk Haskell, který nám poslouží jako prototyp funkcionálního jazyka, jenž je vhodný pro demonstraci problematiky specifikace funkce *If-then-else*), kde je základním prvkem programu *výraz*, ne příkaz, jak je tomu u imperativních jazyků. Jednodušší sémantika funkcionálních jazyků nám také umožní lépe porozumět problematice spjaté se specifikací funkce *If-then-else*.

4.2.1 Striktnost a non-striktnost

Funkce f je striktní na argumentu x, jeli $f(x/\perp) = \perp$. Neplatí-li předcházející tvrzení, je f na parametru x non-striktní [7][8].

4.2.2 Výraz

Termín *výraz* v programování koliduje s *výrazem* z oblasti logické analýzy přirozeného jazyka. V prvním případě je to posloupnost proměnných, konstant a aplikací funkcí (případně dalších pomocných symbolů, například „(“ a „)“), je tedy podobný *výrazu* matematickému (například `(5 * 8 - 9) `mod` 7)`) [9]. Narazíme-li na *výraz* při logické analýze přirozeného jazyka, a tedy i pokud se bavíme o TIL, je jím myšlen *jazykový výraz*, řetězec znaků mající nějaký význam (například „Slunce svítí“). *Výraz* v programování je v logické analýze přirozeného jazyka pouze posloupností znaků, tedy *jazykovým výrazem*, takovým *výrazům* v TIL přiřazujeme jakožto jejich význam určitou proceduru, konstrukci, tak jak byla definována v kapitole 2, v sekci 3.1.4.

V této části kapitoly budeme používat termín *výraz* ve významu, jaký je mu přikládán v programování, je ovšem důležité si uvědomit, že vztáhneme-li tuto problematiku na TIL, jedná se o *Konstrukci*.

[13] If **P** then $\perp$ else A = $\perp$; If **P** then $\perp$ else $\perp$ = $\perp$; If **N** then A else $\perp$ = $\perp$; If **N** then $\perp$ else $\perp$ = $\perp$; If $\perp$ then A else B = $\perp$; If $\perp$ then A else $\perp$ = $\perp$; If $\perp$ then $\perp$ else A = $\perp$; If $\perp$ then $\perp$ else $\perp$ = $\perp$.

4.2.3 Redukční krok a redukční strategie

Mějme funkci $f\ x_1 \ldots x_n = N$[14] a *výraz* $f\ M_1 \ldots M_n$ (aplikace této funkce na argumenty – obecně nějaké *výrazy*). Redukční krok je potom úprava *výrazu*, při níž se některý jeho *podvýraz* tvaru $f\ M_1 \ldots M_n$, nahradí pravou stranou definice funkce, tj. *výrazem* N, a přitom je každý výskyt proměnné x_i nahrazen odpovídajícím výrazem M_i. Redukčním krokem rozumíme i takové transformace *výrazu*, kdy se vyčíslí jednoduché (aritmetické, logické, ...) operace. Skutečnost, že se výraz P v jednom redukčním kroku transformuje na výraz Q zapisujeme $P \rightsquigarrow Q$. Například `fact(3 * 2)` $\rightsquigarrow$ `fact(6)`.

Redukční strategie je pravidlo, které určuje pro každý *výraz* první redukční krok [10].

4.2.4 Striktní vs. líná redukční strategie

Používá-li programovací jazyk striktní redukční strategii, nejprve se zjednodušují argumenty funkcí, tj. ve *výrazu* $M\ N_1 \ldots N_n$ se nejprve redukuje podvýraz N_n, pokud nelze redukovat, redukuje se N_{n-1}, pokud se rovněž nedá redukovat, redukuje se N_{n-2} atd. Pokud nelze redukovat ani jeden z *výrazů* $N_1, \ldots, N_n$, redukuje se výraz M. Konečně, pokud se ani ten nedá redukovat, je M funkcí a celý *výraz* $M\ N_1 \ldots N_n$ se nahradí pravou stranou její definice. Argumenty funkcí se při použití striktní redukční strategie vyhodnocují právě jednou. Striktní redukční strategii se někdy též říká volání hodnotou (call by value).

Chování líné (lazy) redukční strategie ilustrujme opět na obecném příkladu *výrazu* $M\ N_1 \ldots N_n$. Nejprve se redukuje *podvýraz* M, pokud redukovat nelze, musí být M funkcí a celý *výraz* $M\ N_1 \ldots N_n$ se nahradí pravou stranou její definice, přičemž se nahradí všechny výskyty formálních parametrů této funkce za *výrazy* N_i. Při násobném dosazení si však tato redukční strategie uloží informaci o tom, které *podvýrazy* jsou stejné (vzniklé dosazením za různé výskyty téže proměnné), v případě jejich dalšího vyhodnocování se redukují pouze jednou (a za všechny výskyty je dosazena výsledná hodnota redukce). Při použití líné redukční strategie se argumenty funkcí vyhodnocují nejvýše jedenkrát. Líné redukční strategii se někdy též říká volání při potřebě (call by need) [10][11].

Příklad 4.2
Pro účely ilustrace fungování striktní a líné redukční strategie definujme následující uživatelsky definované funkce cube a ifThenElse v Haskellu.

```
cube :: Int -> Int
cube x = x * x * x

ifThenElse :: Bool -> t -> t -> t
ifThenElse p a b = case p of
                     True -> a
                     False -> b
```

[14] Použitá notace vychází z jazyka Haskell, argumenty jsou odděleny mezerou a na prvním místě je jméno funkce (prefixní zápis).

Postup líného vyhodnocování

- `cube (3 * 4)` $\rightsquigarrow$ `(3 * 4) * (3 * 4) * (3 * 4)` $\rightsquigarrow$ `12 * 12 * 12` $\rightsquigarrow$ `144 * 12` $\rightsquigarrow$ `1728`
- `ifThenElse (1 = 1) 1 (1 'div' 0)` $\rightsquigarrow$ `case (1 = 1) of True -> 1 False -> (1 'div' 0)` $\rightsquigarrow^*$ `1`

Poznámka 4.3 „$\rightsquigarrow^*$“ je tranzitivním uzávěrem relace redukčního kroku „$\rightsquigarrow$“. Zápis $P \rightsquigarrow^* Q$ znamená, že po konečně mnoha krocích se P zredukuje na Q.

Implementační detaily case-*výrazu* v Haskellu jsou uživatelům jazyka skryty, proto nejsou ukázány všechny redukční kroky, které vedou až k výsledku (v tomto případě 1).

Postup striktního vyhodnocování

- `cube (3 * 4)` $\rightsquigarrow$ `cube 12` $\rightsquigarrow$ `12 * 12 * 12` $\rightsquigarrow$ `144 * 12` $\rightsquigarrow$ `1728`
- `ifThenElse (1 = 1) 1 (1 'div' 0)` $\rightarrow$ `ERROR`

Striktní redukční strategie narazí na výraz `(1 'div' 0)` již v prvním redukčním kroku, pokus o vyhodnocení tohoto výrazu skončí chybou, nulou nelze dělit.

Poznámka 4.4 Jazyk Haskell má non-striktní sémantiku, tedy i líné vyhodnocování, předchozí ukázka striktní evaluace je čistě hypotetická a ukazuje, jak by se postupovalo, kdyby se uplatnila striktní redukční strategie.

■

Specifikace funkce *If-then-else* s ohledem na požadavky, které jsme formulovali v sekci 4.1, je problematická, protože je v rozporu se striktní evaluační (= redukční) strategií. Ta před vstupem do těla funkce vždy vyhodnotí všechny argumenty funkce, proto je-li některý z nich nedefinován, celá funkce selže (vyhodí se výjimka nebo je vyvolána jiná adekvátní reakce). Proto v jazycích se striktní evaluací nelze na uživatelské úrovni podobné funkce definovat, na rozdíl od jazyků používajících například lazy evaluační strategii, kde je definování non-striktních funkcí v pořádku.

4.3 *If-then-else* v TIL

4.3.1 Striktní definice

V TIL platí tzv. princip kompozicionality (vysvětleno v kapitole 2), je to myšlenka velmi podobná striktnosti z teorie programovacích jazyků (4.2.1). Jednoduchá úvaha vedoucí k následující definici je chybná, neboť nevlastní konstituent činí celou konstrukci, která tento konstituent obsahuje, nevlastní (propagace parciality nahoru). Podobné chování jsme pozorovali při dosazení nedefinovaných *výrazů* za parametry non-striktní funkce v jazyce se striktní evaluací (příklad 4.2).

Specifikace *If-then-else* je čerpána z [2], stejně jako hlavní myšlenky stojící za textem v této části kapitoly.

Příklad 4.3
Pokus o definici *If-then-else*:

$${}^0\textit{If-then-else} = \lambda p\, d_1\, d_2\, [{}^0I^{\alpha}\ \lambda d\, [p \wedge [d = d_1]] \vee [\neg p \wedge [d = d_2]]]$$

Typy: $p/*_n \to^v o$; d, d_1, $d_2/*_n \to^v \alpha$; *If-then-else*$/(\alpha o \alpha \alpha)$.

Jakmile chceme použít tuto definici na příklad níže, objeví se onen zmíněný problém s nevlastní konstrukcí, kvůli které je celá *Kompozice* nevlastní.

$[{}^0\textit{If-then-else}\ {}^0\mathrm{P}\ {}^0 1\ [{}^0\textit{Div}\ {}^0 1\ {}^0 0]]$ ∎

Řešení tohoto problému je možné díky *Trivializaci* a hyperintenzionalitě. Pokud totiž budou *proměnné* z předcházejícího příkladu d, d_1 a d_2 v-konstruovat konstrukce a kýžené argumenty trivializujeme, tedy místo $[{}^0\textit{Div}\ {}^0 1\ {}^0 0]$ použijeme ${}^0[{}^0\textit{Div}\ {}^0 1\ {}^0 0]$, budou zde mít hyperintenzionální výskyt a díky *Trivializaci* nemohou být nevlastní. Nyní probíhá vyhodnocení funkce ve dvou fázích, nejprve se vybere na základě podmínky p jedna z procedur d_1, d_2 a poté se tato procedura provede. *Trivializace* funkčních argumentů zvyšuje úroveň abstrakce na hyperintenzionální úroveň, proto se vybraná konstrukce musí provést ještě jednou, využije se tedy *Dvojího provedení*.

Definice 4.1 Funkce *If-then-else*

$${}^0\textit{If-then-else} = \lambda p\, d_1\, d_2\, {}^2[{}^0I^{*}\ \lambda c\, [p\ \wedge\ [c = d_1]]\ \vee\ [\neg p\ \wedge\ [c = d_2]]]$$

Typy: $p/*_n \to^v o$; c, d_1, $d_2/*_{n+1} \to^v *_n$; ${}^2c$, ${}^2d_1$, ${}^2d_2 \to \alpha$; *If-then-else*$/(\alpha o *_n *_n)$.

Definice funkce *If-then-else* se může zdát na první pohled poněkud komplikovaná. Je důležité si uvědomit, že každý *Uzávěr* určuje charakteristickou funkci nějaké množiny, která vrátí pravdivostní hodnotu podle toho, zda argument je či není prvkem dané množiny. *Uzávěr* $\lambda c\, [p \wedge [c = d_1]] \vee [\neg p \wedge [c = d_2]]$ konstruuje (jednoprvkovou) množinu konstrukcí a její charakteristická funkce vrátí pravdivostní hodnotu **P** právě tehdy, když p v-konstruuje **P** a argument c je roven konstrukci d_1 nebo pokud p v-konstruuje **N** a argument je shodný s konstrukcí d_2. ${}^0I^{*}$ je *Singularizátor* nad množinou konstrukcí (popsán v 2.1.10), vrátí jediný prvek dané jednoprvkové množiny (jinak selže a *Kompozice* je tím pádem nevlastní). Výstupem prvního *Provedení* je tedy konstrukce, která ovšem není definitivním výsledkem, je nutné ji provést ještě jednou (hyperintenzionální úroveň).

Pro větší přehlednost a srozumitelnost budeme aplikaci funkce *If-then-else* zapisovat přirozeněji ve tvaru If C then A else B místo běžného TILovského prefixního zápisu.

4.3.2 Využití

Funkce *If-then-else* je hojně využívána v oblasti analýzy vět spojených s presupozicí. Zprávy obsahující presupozice se mohou objevit například v multiagentních systémech, kde je každý agent samostatnou jednotkou (strojem), která se chová v jistém smyslu

inteligentně. Na konci kapitoly bude prezentována ukázka takovéto komunikace několika inteligentních agentů 4.4. Nejprve je však nutné pokrýt teoretický základ, který stojí za fungováním této aplikace funkce *If-then-else* [12][13].

Presupozice Odpovědět na některé typy otázek typu ano/ne není možné, pokud není splněn určitý předpoklad. Tento předpoklad se nazývá presupozice. Smysluplnou odpověď na otázku spojenou s presupozicí, tedy ano či ne v závislosti na tom, zda je propozice určená touto otázkou pravdivá či nikoliv, nelze dát v případě, že je presupozice nepravdivá. Taková odpověď by byla nesmyslná. Zmíněná propozice tedy nemá v daném $\langle w, t\rangle$, kdy je presupozice nepravdivá, žádnou pravdivostní hodnotu.

Příklad 4.4
Odpověď na větu „Už jste přestal kouřit?“ nemůže být kladná ani záporná, pokud dotyčný nikdy nekouřil. Kdyby odpověděl ano, znamenalo by to, že již někdy kouřil, v opačném případě se přiznává k tomu, že dosud kouří. Presupozicí je v tomto případě propozice, že tázaný někdy v minulosti kouřil. ■

Definice 4.2 Presupozice
Nechť P, $Q \to o_{\tau\omega}$ jsou konstrukce propozic. Pak Q je **presupozicí** *P právě když $(P \models Q)$ a $(\lambda w \lambda t\, \neg P_{wt} \models Q)$.*

Poznámka 4.5 Q vyplývá z P i non-P. Důsledkem tedy je, že pokud non-Q je pravda, pak P nemá žádnou pravdivostní hodnotu (P nemůže být pravdivá i nepravdivá zároveň).

Aplikace Cílem našeho snažení je schopnost pro každou propozici určit její pravdivostní hodnotu v daném $\langle w, t\rangle$ a nebo nevrátit žádnou pravdivostní hodnotu, pokud není příslušná presupozice splněna. S tímto úkolem nám pomůže modifikovaná varianta funkce *If-then-else*, tj. *If-then-else-fail*.

Definice 4.3 Funkce *If-then-else-fail*

$$^0\textit{If-then-else-fail} = \lambda p\, d_1\, {}^2[{}^0I^*\, \lambda c\, [p \wedge [c = d_1]]]$$

Typy: $p/*_n \to^v o$; $c, d_1/*_{n+1} \to^v *_n$; ^2c, $^2d_1 \to \alpha$; *If-then-else-fail*$/(\alpha o *_n)$.

V případě, že parametr p v-konstruuje **N**, celá *Kompozice* selhává, je nevlastní. To však naprosto vyhovuje naší potřebě, dosadíme-li za p presupozici a za d_1 s ní spojené tvrzení (propozici).

Obdobně jako u funkce *If-then-else* i v případě *If-then-else-fail* budeme používat notační zjednodušení ve tvaru If C then A else fail.

Příklad 4.5
Tento příklad je modifikací části ukázkové komunikace agentů 4.4.2.

$A \to B$: Dojel jsi do servisního centra v Ostravě?[15]

$\lambda w \lambda t$ [If [^{0}*Jede*$_{wt}$ ^0B ^{0}*OSC*] then [$\lambda w \lambda t$ [^{0}*Dojet*$_{wt}$ ^0B ^{0}*OSC*]] else *fail*]

Typy: *B*, *OSC*/ι; *Dojet*, *Jede*/$(o\iota\iota)_{\tau\omega}$. ■

Věty v minulém a budoucím čase Ze všech vět spojených s presupozicí můžeme vyčlenit ty, které jsou v minulém nebo budoucím čase. Presupozice těchto vět bývají často poněkud komplikovanější, chceme-li vyjádřit například frekvenci výskytu nějakého jevu nebo mluvíme o určitém časovém intervalu (den, týden, . . .). Pro tyto účely bylo navrženo obecné schéma analýzy časových vět, které bylo představeno v [14]. Pro demonstraci nám poslouží následující věty, z jejichž analýzy poté vyplyne ono schéma.

1. Jan byl po celý rok 2013 v zahraničí.

   - Zde je presupozicí to, že rok 2013 již uplynul. Nemůžeme vědět, jestli se Jan nevrátí do své země, pokud rok 2013 ještě trvá.
   - $\lambda w \lambda t$ [If $\forall t_1$ [[^{0}*Rok13* t_1] $\supset$ [$t_1 < t$]] then $\forall t'$ [[^{0}*Rok13* t'] $\supset$ [^{0}*Být_v_zahraničí*$_{wt'}$ ^{0}*Jan*]] else *fail*]

2. Jan byl v roce 2013 dvakrát v zahraničí.

   - Presupozice je stejná jako u předchozí věty. Zde ukážeme, jak se vypořádat s frekvencí výskytu událostí.
   - $\lambda w \lambda t$ [If $\forall t_1$ [[^{0}*Rok13* t_1] $\supset$ [$t_1 < t$]] then [[^{0}*Dvakrát*$_w$ $\lambda w \lambda t$ [^{0}*Být_v_zahraničí*$_{wt}$ ^{0}*Jan*]] ^{0}*Rok13*] else *fail*]
   - [[^{0}*Dvakrát*$_w$ $\lambda w \lambda t$ [^{0}*Být_v_zahraničí*$_{wt}$ ^{0}*Jan*]] ^{0}*Rok13*] = [^{0}*Card* λd [$\forall t$ [[d t] $\supset$ [^{0}*Být_v_zahraničí*$_{wt}$ ^{0}*Jan*]] $\wedge \exists t$ [[d t] $\wedge$ [^{0}*Rok13* t]]] = ^{0}2]
   - *Card* vrátí kardinalitu dané množiny. Proměnná d nabývá hodnot disjunktních souvislých časových intervalů.

3. Jan bude 20. 12. 2014 celý den v zahraničí.

   - Presupozice pro tuto větu na rozdíl od té předchozí předpokládá, že zmiňovaný den 20. 12. 2014 ještě nenastal, protože věta hovoří o budoucnosti.
   - $\lambda w \lambda t$ [If $\forall t_1$ [[^{0}*20_12_2014* t_1] $\supset$ [$t_1 > t$]] then $\forall t'$ [[^{0}*20_12_2014* t'] $\supset$ [^{0}*Být_v_zahraničí*$_{wt'}$ ^{0}*Jan*]] else *fail*]

Typy: *Rok13*, *20_12_2014*/$(o\tau)$; *Být_v_zahraničí*/$(o\iota)_{\tau\omega}$; *Dvakrát*/$((o(o\tau))o_{\tau\omega})_\omega$; *Card*/$(\tau(o\alpha))$; $d \to (o\tau)$.

[15] Presupozice spojená s touto větou je propozice, že *B* jede do ostravského servisního centra.

Poznámka 4.6 Uvažujeme pouze věty v minulém čase, jež v sobě zahrnují konkrétní čas, kdy popisovaná událost nastala (bez tohoto údaje jsou věty poněkud nekompletní, nabízí se otázka, kdy daná událost proběhla). Věty, které nenesou časové určení můžeme také analyzovat, avšak v jejich případě odpadá problém s presupozicí, nejsou pro nás tedy zajímavé.

V první vzorové větě je presupozice vyjádřena konstrukcí $\forall t_1\,[[^0\mathit{Rok13}\;t_1] \supset [t_1 < t]]$. Chceme docílit toho, aby nebyla vrácena žádná pravdivostní hodnota (selhání) v případě, že časový interval, o kterém se ve větě mluví (rok 2013) nepředchází aktuálnímu času, přesněji tedy pokud pro všechny časové okamžiky t_1 platí, že pokud jsou z intervalu určeného rokem 2013, pak všechny předcházejí aktuálnímu času t ($t_1 < t$).

Ve druhé vzorové větě je presupozice shodná s tou z věty první. Zajímavý je zde modifikátor frekvence *Dvakrát*, jehož denotátem je funkce závislá na světě, která bere jako argument propozici. Vrátí funkci, která na základě parametru, který představuje referenční interval – časový úsek, o němž se ve větě hovoří (v našem případě rok 2013), rozhodne zda tento interval patří do množiny všech intervalů, ve kterých byla daná propozice pravdivá (právě) dvakrát (průnik chronologie propozice a referenčního intervalu má kardinalitu 2).

Presupozice třetí věty je analogií dvou předešlých. Tentokrát však vyžadujeme, aby aktuální čas předcházel časovému údaji, o němž se ve větě hovoří (20. 12. 2014). Nyní na základě těchto pozorování můžeme zformulovat obecné schéma pro analýzu vět v minulém a budoucím čase. Funkce *If-then-else-fail* bude pro zkrácení analýzy zahrnuta již v definici schématu, nebude ji tedy třeba explicitně použít.

Zaveďme tyto dva složené typy a dvě funkce:

- $\overline{\text{V\_čase}}$[16] nazveme typ $(o\tau)$,
- $\overline{\text{Frekvence}}$ nazveme typ $((o(o\tau))o_{\tau\omega})_\omega$,
- *Minulost*$/(o(o(o\tau))(o\tau))_\tau$,
- $\leq_\tau/(o(o\tau)\tau)$.

Označme větu (vyjádřenou propozicí) v minulém čase M, jejíž referenční interval bude typu $\overline{\text{V\_čase}}$. Operátor $\leq_\tau$ vrací **P**, předchází-li (nebo je roven) daný časový interval danému časovému okamžiku, v opačném případě **N**. Funkce typu $\overline{\text{Frekvence}}$ slouží k vyjádření frekvence výskytu události (pravdivosti propozice v určitém počtu časových okamžiků, příkladem může být funkce *Dvakrát* použitá ve druhé vzorové větě). Výstupem takové funkce je množina disjunktních časových intervalů, ve kterých je daná propozice v daném světě pravdivá. Funkce *Minulost* v závislosti na čase vyhodnocení, referenčním intervalu (propozice) typu $\overline{\text{V\_čase}}$ a množině intervalů (ve kterých je propozice pravdivá – výstup funkce typu $\overline{\text{Frekvence}}$) vrací **P**, **N** podle toho, zda daný referenční interval

[16] Pro odlišení skutečnosti, že se jedná o pojmenovaný typ (a sám o sobě není součástí konstrukcí), budeme jejich názvy značit „nadtržením“.

patří do množiny intervalů či nikoliv, v případě, že času vyhodnocení nepředchází celý referenční interval, funkce selhává.

$\lambda w\lambda t\,[{}^0\mathit{Minulost}\,[{}^0\mathit{Frekvence}_w\ M]\ {}^0V\_\mathit{case}] = \lambda w\lambda t\,[\text{If}\,[{}^0V\_\mathit{case} \leq_\tau t]\ \text{then}\,[[{}^0\mathit{Frekvence}_w\ M]\ {}^0V\_\mathit{case}]\ \text{else}\ \mathit{fail}]$

Poznámka 4.7 $\overline{\text{Frekvence}}$ a $\overline{\text{V\_čase}}$ při aplikaci nahradíme konkrétními funkcemi (jak vidno z příkladu níže).

Analýza první vzorové věty s využitím tohoto schématu vypadá takto:

$\lambda w\lambda t\,[{}^0\mathit{Minulost}\,[{}^0\mathit{Celý}_w\ \lambda w\lambda t\,[{}^0\mathit{Být\_v\_zahraničí}_{wt}\ {}^0\mathit{Jan}]]\ {}^0\mathit{Rok13}]$.

Uvedené schéma je možno použít pro věty v minulém čase. Podobné schéma pro věty v čase budoucím získáme drobnou modifikací již uvedeného schématu.

Potřebujeme definovat ještě tyto funkce:

- $\mathit{Budoucnost}/(o(o(o\tau))(o\tau))_\tau$,
- $\geq_\tau/(o(o\tau)\tau)$.

Operátor $\geq_\tau$ funguje opačně vůči $\leq_\tau$, vrátí **P**, pokud daný časový interval následuje po (nebo je roven) daném časovém okamžiku, jinak **N**. Funkce *Budoucnost* se liší od funkce *Minulost* pouze v tom, že využívá operátoru $\geq_\tau$.

$\lambda w\lambda t\,[{}^0\mathit{Budoucnost}\,[{}^0\mathit{Frekvence}_w\,M]\,{}^0V\_\mathit{case}] = \lambda w\lambda t\,[\text{If}\,[{}^0V\_\mathit{case} \geq_\tau t]\ \text{then}\,[[{}^0\mathit{Frekvence}_w\,M]\ {}^0V\_\mathit{case}]\ \text{else}\ \mathit{fail}]$

Analýza třetí vzorové věty s využitím tohoto schématu vypadá takto:

$\lambda w\lambda t\,[{}^0\mathit{Budoucnost}\,[{}^0\mathit{Celý}_w\ \lambda w\lambda t\,[{}^0\mathit{Být\_v\_zahraničí}_{wt}\ {}^0\mathit{Jan}]]\ {}^0\mathit{20\_12\_2014}]$.

4.4 Komunikace agentů

4.4.1 Obecné schéma posílaných zpráv

V komunikaci agentů využijeme tzv. message performatives (česky by se to dalo označit jako „obálky“ zpráv). Jedná se o informaci o typu zprávy – zda se jedná o otázku, oznámení či příkaz. Je to v podstatě strojové vyjádření typu věty (rozkazovací, oznamovací, tázací). Obálek je v komunikaci agentů potřeba, jelikož obsah každé zprávy označuje nějakou propozici a nebylo by tak jasné, zda se jedná o otázku, příkaz nebo oznámení.

V TIL se modeluje komunikace agentů pomocí schématu, které jasně určí, o jaký typ zprávy se jedná, od koho je a komu byla určena, součástí je i samotná zpráva [13].

$\lambda w\lambda t\,[{}^0\mathit{Message}_{wt}\ {}^0\mathit{Who}\ {}^0\mathit{Whom}\ \lambda w\lambda t\,[{}^0\mathit{Kind}_{wt}\ \mathit{What}]]$

Typy: *Message*/$(o\iota\iota o_{\tau\omega})_{\tau\omega}$; $\overline{\text{Kind}}$ = $(oo_{\tau\omega})_{\tau\omega}$; *Query*, *Inform*, *Order*/$\overline{\text{Kind}}$; *Who*, *Whom*/ι; *What* $\rightarrow o_{\tau\omega}$.

Za *Who* a *Whom* se dosadí příslušná individua, po řadě odesílatel a adresát, $\overline{\text{Kind}}$ je typ (typu) zprávy, může to být buď *Query* (otázka), *Inform* (odpověď, oznámení) nebo *Order* (příkaz). *What* je samotná zpráva, konstrukce konstruující propozici. Použití uvedeného schématu je vidět dále na příkladu komunikace agentů. . .

4.4.2 Demonstrace komunikace agentů

Agenti jsou samostatné mobilní stroje komunikující mezi sebou a sbírající informace o okolí, které se poté ukládají do znalostní báze systému, kde mohou být využity k nejrůznějším účelům. Následující příklad přibližuje situaci, kdy je potřeba zkontrolovat technický stav agentů v servisu.

Agenti *A*, *B*, *C* jsou běžnými agenty, tak jsou popsáni výše. Agent *D* je nadřazen agentům *A*, *B* a *C* v tom smyslu, že jim zadává příkazy, je to jejich lokální dispečer a je jim tedy nadřazen. Má rozšířenou bázi znalostí, která je pravidelně doplňována ostatními agenty, proto je schopen odpovídat i na komplexnější empirické dotazy. *D* má přehled o aktuální pozici (GPS) všech ostatních agentů.

Centrální agent *D* vyžaduje kontrolu agentů *A*, *B* a *C* v ostravském servisním centru, a za tímto účelem jim posílá zprávu. Agentům *A* a *B* je zpráva doručena, avšak agent *C* se nachází v prostředí, kde je bezdrátové připojení ke komunikační síti vysoce nestabilní, příkaz se k němu tedy nedostane.

1. $D \rightarrow \{A, B, C\}$ (8.50): [ORDER] Přijeďte na 11.00 na technickou prohlídku do servisního centra v Ostravě.
   - Doručeno – *A*: 8.51, *B*: 8.52.
   - Nedoručeno (do 15 minut) – *C*.
     - $D \rightarrow A$ (8.50):
       $\lambda w\lambda t[{}^0\mathit{Message}_{wt}\ {}^0D\ {}^0A\ \lambda w\lambda t\,[{}^0\mathit{Order}_{wt}$
       $[\lambda w\lambda t\,[[[{}^0\mathit{Přijet}\ w]\ {}^0\mathit{Čas\_11\_00}]\ {}^0A\ {}^0\mathit{OSC}]]$
       $\wedge\,[\lambda w\lambda t\,[[[{}^0\mathit{Přijet}\ w]\ {}^0\mathit{Čas\_11\_00}]\ {}^0B\ {}^0\mathit{OSC}]]$
       $\wedge\,[\lambda w\lambda t\,[[[{}^0\mathit{Přijet}\ w]\ {}^0\mathit{Čas\_11\_00}]\ {}^0C\ {}^0\mathit{OSC}]]]]$
     - $D \rightarrow B$ (8.50):
       $\lambda w\lambda t[{}^0\mathit{Message}_{wt}\ {}^0D\ {}^0B\ \lambda w\lambda t\,[{}^0\mathit{Order}_{wt}$
       $[\lambda w\lambda t\,[[[{}^0\mathit{Přijet}\ w]\ {}^0\mathit{Čas\_11\_00}]\ {}^0A\ {}^0\mathit{OSC}]]$
       $\wedge\,[\lambda w\lambda t\,[[[{}^0\mathit{Přijet}\ w]\ {}^0\mathit{Čas\_11\_00}]\ {}^0B\ {}^0\mathit{OSC}]]$
       $\wedge\,[\lambda w\lambda t\,[[[{}^0\mathit{Přijet}\ w]\ {}^0\mathit{Čas\_11\_00}]\ {}^0C\ {}^0\mathit{OSC}]]]]$
     - $D \rightarrow C$ (8.50):
       $\lambda w\lambda t[{}^0\mathit{Message}_{wt}\ {}^0D\ {}^0C\ \lambda w\lambda t\,[{}^0\mathit{Order}_{wt}$
       $[\lambda w\lambda t\,[[[{}^0\mathit{Přijet}\ w]\ {}^0\mathit{Čas\_11\_00}]\ {}^0A\ {}^0\mathit{OSC}]]$

$\wedge\,[\lambda w\lambda t\,[[[{}^0\textit{Přijet}\ w]\ {}^0\textit{Čas\_11\_00}]\ {}^0B\ {}^0OSC]]$
$\wedge\,[\lambda w\lambda t\,[[[{}^0\textit{Přijet}\ w]\ {}^0\textit{Čas\_11\_00}]\ {}^0C\ {}^0OSC]]]]$

- Typy: $\textit{Přijet}/(o\iota\iota)_{\tau\omega}$; A, B, C, D, OSC/ι; $\textit{Čas\_11\_00}/\tau$.

Poznámka 4.8 Zprávy obsahují informaci o tom, kteří (další) agenti se mají účastnit prohlídky v ostravském servisním centru, díky tomu o sobě agenti „vědí“. To bude důležité později, kdy se agenti budou navzájem kontrolovat.

2. $A{\rightarrow}D$ (8.52): [INFORM] Ano, přijedu...

   - Doručeno – D: 8.53.
     - $A{\rightarrow}D$ (8.52):
       $\lambda w\lambda t\,[\text{If}\ [t < {}^0\textit{Čas\_11\_00}]$
       $\text{then}\ [{}^0\textit{Message}_{wt}\ {}^0A\ {}^0D\ \lambda w\lambda t\,[{}^0\textit{Inform}_{wt}$
       $\lambda w\lambda t\,[[[{}^0\textit{Přijet}\ w]\ {}^0\textit{Čas\_11\_00}]\ {}^0A\ {}^0OSC]]]$
       $\text{else}\ [{}^0\textit{Message}_{wt}\ {}^0A\ {}^0D\ \lambda w\lambda t\,[{}^0\textit{Inform}_{wt}\ \lambda w\lambda t\,\neg[t < {}^0\textit{Čas\_11\_00}]]]]$

3. $B{\rightarrow}D$ (8.52): [INFORM] Ano, přijedu...

   - Doručeno – D: 8.53.
     - $B{\rightarrow}D$ (8.52):
       $\lambda w\lambda t\,[\text{If}\ [t < {}^0\textit{Čas\_11\_00}]$
       $\text{then}\ [{}^0\textit{Message}_{wt}\ {}^0B\ {}^0D\ \lambda w\lambda t\,[{}^0\textit{Inform}_{wt}$
       $\lambda w\lambda t\,[[[{}^0\textit{Přijet}\ w]\ {}^0\textit{Čas\_11\_00}]\ {}^0B\ {}^0OSC]]]$
       $\text{else}\ [{}^0\textit{Message}_{wt}\ {}^0B\ {}^0D\ \lambda w\lambda t\,[{}^0\textit{Inform}_{wt}\ \lambda w\lambda t\,\neg[t < {}^0\textit{Čas\_11\_00}]]]]$

Poznámka 4.9 Jakmile je zpráva doručena, musí si agenti zkontrolovat, zda je presupozice zprávy pravdivá nebo ne a na základě toho poslat zpět adekvátní odpověď. V případě, že je splněna, pošle adresát zpátky patřičnou odpověď, jinak pošle zpátky negovanou presupozici, aby odesílatele informoval o nastalé (mimořádné) situaci.

D zkouší znovu kontaktovat C každých 15 minut, nicméně zpráva k němu nikdy nedorazí. Pro jednoduchost agent D nepodniká žádné další kroky, aby agenta C zkontaktoval, mohl by například vyslat dalšího agenta za agentem C, aby ho zkontaktoval „osobně“. Takové řešení však příkladu nedá žádnou další přidanou hodnotu.

4. $D{\rightarrow}C$ (9.05 + $k \cdot$ 15 minut): [ORDER] Přijeď na 11.00 na technickou prohlídku do servisního centra v Ostravě.[17]

   - Nedoručeno (do 15 minut) – C.
     - (zpráva se opakuje)

Agenti si po tom, co dorazí na místo určení posílají dotazy, zda i ostatní již dorazili. Jedná se o určitou vzájemnou kontrolu. Každý agent se stará o jiného agenta, v tomto případě se agent A stará o B, B kontroluje C a C dohlíží na A.

[17] Proměnná k postupně nabývá hodnot 1, 2, ..., dokud se nepodaří zprávu doručit.

5. $A \rightarrow B$ (10.20): [QUERY] Dojel jsi do servisního centra v Ostravě?

   - Doručeno – B: 10.21.
     - $A \rightarrow B$ (10.20):
       $\lambda w \lambda t\, [^0\mathit{Message}_{wt}\ ^0A\ ^0B\ \lambda w \lambda t\, [^0\mathit{Query}_{wt}\ \lambda w \lambda t\, [^0\mathit{Přijet}_{wt}\ ^0B\ ^0\mathit{OSC}]]]$

6. $B \rightarrow A$ (10.21): [INFORM] Nedojel, jsem na cestě.

   - Doručeno – A: 10.22.
     - $B \rightarrow A$ (10.21):
       $\lambda w \lambda t$ [If $[^0\mathit{Jede}_{wt}\ ^0B\ ^0\mathit{OSC}]$
       then $[^0\mathit{Message}_{wt}\ ^0B\ ^0A\ \lambda w \lambda t\, [^0\mathit{Inform}_{wt}\ \lambda w \lambda t\, \neg[^0\mathit{Přijet}_{wt}\ ^0B\ ^0\mathit{OSC}]]]$
       else $[^0\mathit{Message}_{wt}\ ^0B\ ^0A\ \lambda w \lambda t\, [^0\mathit{Inform}_{wt}\ \lambda w \lambda t\, \neg[^0\mathit{Jede}_{wt}\ ^0B\ ^0\mathit{OSC}]]]$
     - Typy: $\mathit{Jede}/(o\iota\iota)_{\tau\omega}$.

Agent B dorazí na místo a dotáže se C, zda již také dorazil na místo. Ten však stále ještě nedostal zprávu od D, že má přijet na technickou kontrolu, není tedy splněna presupozice, že „jel na určené místo“.

7. $B \rightarrow C$ (10.28): [QUERY] Dojel jsi do servisního centra v Ostravě?

   - Doručeno – C: 10.28.
     - $B \rightarrow C$ (10.28):
       $\lambda w \lambda t\, [^0\mathit{Message}_{wt}\ ^0B\ ^0C\ \lambda w \lambda t\, [^0\mathit{Query}_{wt}\ \lambda w \lambda t\, [^0\mathit{Přijet}_{wt}\ ^0C\ ^0\mathit{OSC}]]]$

8. $C \rightarrow B$ (10.29): [INFORM] Nedojel, nikam jsem nejel.

   - Doručeno – B: 10.30.
     - $C \rightarrow B$ (10.29):
       $\lambda w \lambda t$ [If $[^0\mathit{Jede}_{wt}\ ^0C\ ^0\mathit{OSC}]$
       then $[^0\mathit{Message}_{wt}\ ^0C\ ^0B\ \lambda w \lambda t\, [^0\mathit{Inform}_{wt}\ \lambda w \lambda t\, \neg[^0\mathit{Přijet}_{wt}\ ^0C\ ^0\mathit{OSC}]]]$
       else $[^0\mathit{Message}_{wt}\ ^0C\ ^0B\ \lambda w \lambda t\, [^0\mathit{Inform}_{wt}\ \lambda w \lambda t\, \neg[^0\mathit{Jede}_{wt}\ ^0C\ ^0\mathit{OSC}]]]$

B přepošle původní zprávu od D, kterou se mu nepodařilo doručit. Agent C je však stále v zóně s omezeným přístupem ke komunikační síti; to, že se podařilo předchozí dvě zprávy doručit bylo dílem náhody. D se pořád pokouší poslat svou zprávu agentu C každých 15 minut (. . . , 10.05, 10.20, 10.35, . . .). Nyní se i agent B snaží agentu C opakovaně předat zprávu o technické prohlídce v servisu, ovšem také marně.

9. $B \rightarrow C$ (10.30 + $l \cdot 15$ minut): [ORDER] Přijeď na 11.00 na technickou prohlídku do servisního centra v Ostravě.[18]

   - Nedoručeno (do 15 minut) – C.

[18] Proměnná l postupně nabývá hodnot 1, 2, . . . , dokud se nepodaří zprávu doručit. Agent B zkouší stejný postupu jako agent D.

- (zpráva se opakuje, je stejná jako zpráva 4. a 1., s tím rozdílem, že odesílatelem je v tomto případě *B*)

Agenti *B* i *D* se snaží agentu *C* doručit zprávu o technické prohlídce v 11.00 hodin, avšak nepodaří se jim to, dokud na to není příliš pozdě, tedy po 11. hodině, kdy se agent *C* v rámci svých povinnosti dostal do oblasti s lepším spojením. Presupozice, že je čas schůzky v budoucnosti, tedy není splněna a tak nezbývá agentu *C* nic jiného než sdělit tento fakt agentu *D*. Mezitím jsou agenti *A* i *B* na technické prohlídce v servise, jak bylo naplánováno.

10. *C*→*D* (11.21): [INFORM] 11.00 $<$ *čas doručení zprávy*, nedostal jsem zprávu včas.

    ■ Doručeno – *D*: 11.22.

    - *C*→*D* (11.21):
      $\lambda w\lambda t\,[\text{If}\,[t < {}^0\check{C}as\_11\_00]$
      $\text{then}\,[{}^0Message_{wt}\ {}^0C\ {}^0D\ \lambda w\lambda t\,[{}^0Inform_{wt}$
      $\lambda w\lambda t\,[[[{}^0Přijet\ w]\ {}^0\check{C}as\_11\_00]\ {}^0C\ {}^0OSC]]]$
      $\text{else}\,[{}^0Message_{wt}\ {}^0C\ {}^0D\ \lambda w\lambda t\,[{}^0Inform_{wt}\ \lambda w\lambda t\,\neg[t < {}^0\check{C}as\_11\_00]]]]$

Agent *D* tedy naplánuje prohlídku agenta *C* v jiném čase. Spočítá, za jak dlouho je schopen dorazit do servisu s ohledem na momentální dopravní situaci a polohu agenta *C*, díky ostatním agentům má tyto informace ve své bázi znalostí. Následně agentu *C* pošle zprávu požadující po něm, aby se dostavil v určeném čase.

Poznámka 4.10 Agent *D* nejprve provede dotaz nad svou bázi znalostí a zjistí, jak dlouho potrvá agentu *C* cesta z jeho aktuální polohy za momentální dopravní situace do ostravského servisního centra (s rezervou 15 minut) a vypočtenou hodnotu si uloží (*Nový_čas_servis*), aby ji poté mohl použít ve zprávě agentu *C*.

$$Nový\_čas\_servis = [\lambda w\lambda t\,[[{}^0Doba\_cesty_{wt}\ [{}^0Lokace_{wt}\ {}^0C]\ [{}^0Lokace_{wt}\ {}^0OSC]\ {}^0C] + {}^0\check{C}as\_15\_minut + t]]_{wt}$$

Typy: *Nový_čas_servis*, *Čas_15_minut*/τ; *Doba_cesty*/$(\tau\mu\mu\iota)_{\tau\omega}$; *Lokace*/$(\mu\iota)_{\tau\omega}$[19].

11. *D*→*C* (11.28): [ORDER] Přijeď na *nově určený čas* na technickou prohlídku do servisního centra v Ostravě.

    ■ Doručeno – *C*: 11.29.

    - *D*→*C* (11.28):
      $\lambda w\lambda t\,[{}^0Message_{wt}\ {}^0D\ {}^0C\ \lambda w\lambda t\,[{}^0Order_{wt}$
      $\lambda w\lambda t\,[[[{}^0Přijet\ w]\ {}^0Nový\_čas\_servis]\ {}^0C\ {}^0OSC]]]$

12. *C*→*D* (11.30): [INFORM] Ano, přijedu...

[19] Typ μ reprezentuje GPS souřadnice. Tento typ není přesně definován, slouží jako abstrakce od konkrétního složeného typu, který může být použit pro reprezentaci GPS dat.

- Doručeno – *D*: 11.31.
  - $C \rightarrow D$ (11.30):
    $\lambda w \lambda t$ [If [$t < {}^0$*Nový_čas_servis*]
    then [^{0}*Message*$_{wt}$ ${}^0C$ ${}^0D$ $\lambda w \lambda t$ [^{0}*Inform*$_{wt}$
    $\lambda w \lambda t$ [[[^{0}*Přijet* w] ^{0}*Nový_čas_servis*] ${}^0C$ ^{0}*OSC*]]]
    else [^{0}*Message*$_{wt}$ ${}^0C$ ${}^0D$ $\lambda w \lambda t$ [^{0}*Inform*$_{wt}$ $\lambda w \lambda t$ $\neg$[$t < {}^0$*Nový_čas_servis*]]]]

Poznámka 4.11 V elektronické příloze se nachází celá ukázka komunikace agentů v jazyce TIL-Script, soubor má název *mas-com-example.ts*.

5 Jednoduchý interpret *If-then-else*

Jazyk TIL-Script byl před a během psaní této práce teprve v raném stádiu vývoje (ačkoliv bylo očekáváno, že již bude interpret TIL-Scriptu hotov). K dispozici tak byl pouze parser napsaný v jazyce Python s možností integrace do jazyka Java pomocí Jythonu, což je implementace jazyka Python pro virtuální stroj Javy [16]. Protože neexistovala fungující programová struktura interpretu, byla tvorba modulu pro tento interpret dosti nepraktická, jelikož by bylo nutné tuto strukturu přinejmenším navrhnout a modul implementovat s tím, že nebude funkční, dokud nebude hotov celý interpret. TIL-Script navíc v současné době prochází řadou změn. Z těchto důvodů jsem se rozhodl jít cestou implementace zjednodušeného interpretu TIL-Scriptu pouze pro funkci *If-then-else*, aby se dalo prezentovat použití této funkce v jazyce TIL-Script. Následně využít zkušenosti z práce s existujícím parserem pro návrh řádného interpretu pro TIL-Script.

Interpret funkce *If-then-else* je schopen interpretovat jen podjazyk TIL-Scriptu, kde se z konstrukcí mohou vyskytovat pouze *Kompozice* a *Trivializace*. Pro účely demonstrace byly implementovány některé matematické a logické funkce, odpovídající klíčovým slovům TIL-Scriptu. Zde je jejich soupis:

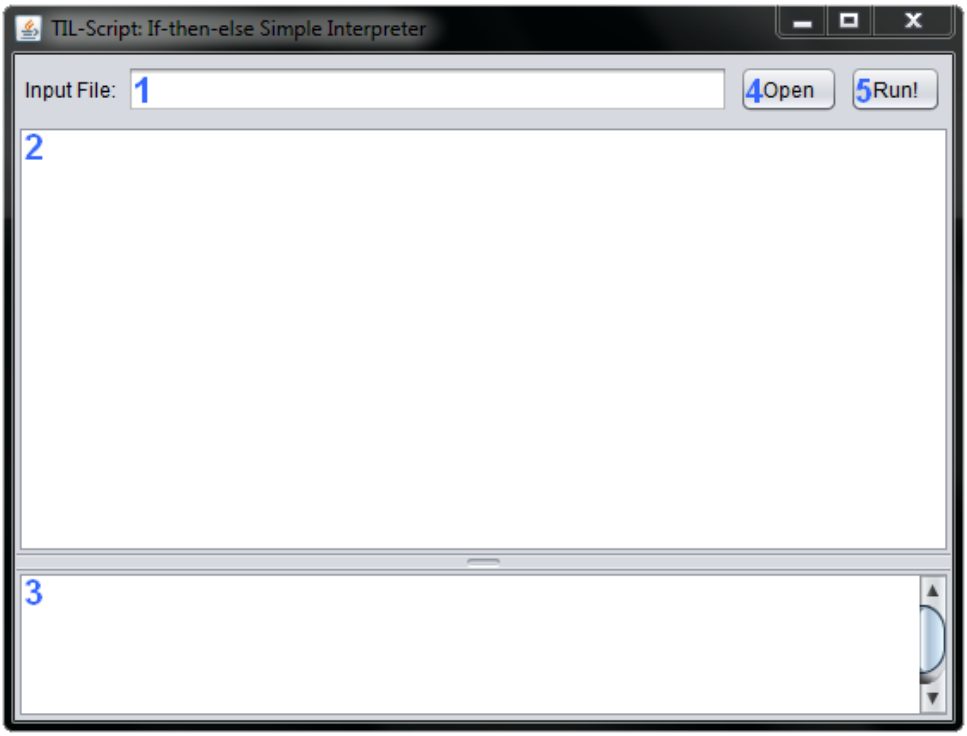
- **+** – funkce sčítání,
- **-** – funkce odčítání,
- * – funkce násobení,
- **/** – funkce dělení,
- **=** – funkce rovnosti celých/reálných čísel,
- **And** – funkce konjunkce,
- **Or** – funkce disjunkce,
- **Implies** – funkce implikace,
- **Not** – funkce negace.

5.1 Popis rozhraní a ovládání

Po spuštění interpretu (dvojklikem na soubor *IfThenElseTILScript.jar*) se zobrazí okno, se kterým uživatel dále pracuje (vizte obrázek **??** s poznámkou 5.1). Nejprve je nutné určit soubor se zdrojovým kódem TIL-Scriptu a to buď „manuálně“, vypsáním absolutní cesty k souboru do textového pole, které je uvedeno popisem „Input File:“, nebo s pomocí dialogu výběru souboru – ten je vyvolán kliknutím na tlačítko „Open“ – po vybrání a potvrzení souboru se jeho obsah načte do hlavního textového pole umístěného v centru okna. Uživatel může, kromě výše popsaných akcí, napsat program rovnou do hlavního textového pole interpretu (a nemusí tedy v tomto případě žádný soubor uvádět). Taktéž je

možné načtený soubor před spuštěním interpretace dále upravovat v hlavním textovém poli interpretu (tyto změny se však nepromítnou do načteného souboru).

Uživatel spustí interpretaci stiskem tlačítka „Run!“. Jakmile je interpretace dokončena, objeví se výsledek ve výstupním textovém poli ve spodní části okna, pokud bylo zadáno více nezávislých výrazů v TIL-Scriptu (ukončených tečkou), jsou výsledky uvozeny postupně dvojicí znaků #1, . . . , #n.



Obrázek 2: Uživatelské rozhraní interpretu

Poznámka 5.1 Pro lepší orientaci v popisu rozhraní jsou v obrázku umístěny číslice identifikující jednotlivé grafické prvky, v seznamu níže jsou tyto prvky pojmenovány a tato jména jsou také v popisu použita.

1. Textové pole pro zadání cesty ke zdrojovému souboru.
2. Hlavní textové pole, kde se zobrazuje zdrojový kód.
3. Výstupní textové pole, zde se zobrazují výsledky interpretace.
4. Tlačítko pro vyvolání dialogu s výběrem kýženého souboru (se zdrojovým kódem).
5. Tlačítko pro spuštění interpretace kódu, který se nachází v hlavním textovém poli.

5.2 Implementace

Interpret je napsán v jazyce Java. Samotné třídy jsou rozděleny do dvou balíčků podle jejich povahy: *main* a *gui*.

V balíčku *main* jsou umístěny třídy, které mají na starost hlavní funkcionalitu interpretu, jmenovitě jsou to třídy **SimpleITEInterpreter** a **TSParser**. Jak je z názvu patrné, třída **SimpleITEInterpreter** skrývá jádro aplikace, samotný interpret. Třída **TSParser** řeší volání parseru na textový vstup – zápis programu v TIL-Scriptu.

Balíček *gui* obsahuje třídy spjaté s uživatelským rozhraním interpretu, které je realizováno s využitím standardní knihovny Swing jazyka Java. Třída **MainWindow** seskupuje prvky uživatelského rozhraní a mapuje možné uživatelské akce (kliknutí na tlačítko apod.) na příslušný obslužný kód. Díky třídě **TSFileFilter**, která rozšiřuje třídu **FileFilter** ze standardní knihovny, uvidí uživatel v dialogu pro výběr zdrojového souboru s programem v TIL-Scriptu pouze soubory s přípustnou příponou. Třída **UnrecognizedEntityException** je třída rozšiřující bázovou třídu výjimek v Javě – **Exception**. Tato výjimka je vyhozena v případě, že interpret narazí v AST (Abstract Syntax Tree) na konstrukci v TIL-Scriptu, kterou není schopen interpretovat (tj. **ClosureNodeInterface**, **VariableNodeInterface** a **NExecutionNodeInterface**).

Do projektu bylo třeba také importovat knihovny *jython.jar* a *myTilscript.jar* a složku s některými potřebnými moduly jazyka Python. Soubor *myTilscript.jar* obsahuje implementaci parseru TIL-Scriptu v Pythonu, Jython umožňuje pracovat s programy v Pythonu a volat je z kódu v Javě.

Informace o třídách, které jsou součástí API jazyka Java pochází z [15].

5.2.1 Balíček *main*

Třída TSParser Tato třída je vytvořena podle návrhového vzoru Singleton, jenž zajišťuje existenci vždy jen jedné instance, se kterou lze pracovat po zavolání statické metody `getInstance()`, jež tuto instanci vrací. Pomocí metody `parse(String code)` dostaneme objekt třídy implementující rozhraní **TSParserType** (součást implementace parseru), zavoláme-li na něm metodu `getAST()` dostaneme seznam kořenových uzlů AST (pro každý výraz v jazyce TIL-Script jeden), s nímž poté pracuje samotný interpret.

Třída SimpleITEInterpreter V této třídě je soustředěna logika interpretu. Nejdůležitější jsou metody `interpretITE(String source)` a `interpret(ASTNodeInterface node)`, třída pak obsahuje ještě další pomocné metody usnadňující práci s uzly AST a metody starající se o aplikaci jednoduchých matematických a logických funkcí.

První z uvedených metod se stará o parsování zdrojového kódu a získání seznamu odpovídajících AST, po ověření platnosti vstupu, tedy *Kompozice*, kde je první konstrukcí *Trivializace* funkce *If-then-else* či *If-then-else-fail*[20]. Následně je spuštěna interpretace jednotlivých vstupů (může jich být více najednou).

[20] Jelikož gramatika nepovoluje v názvech objektů spojovníky, byly zvoleny varianty s podtržítkem, tedy „If_then_else“ a „If_then_else_fail“.

Vlastní interpretaci zajišťuje druhá zmíněná metoda `interpret(ASTNodeInterface node)`. **ASTNodeIndertface** je interface pro práci s uzlem AST, v metodě jsou použita rozšíření tohoto rozhraní, jmenovitě tedy **EntityNodeInterface**, **TrivialisationNodeInterface** a **CompositionNodeInterface**. V závislosti na typu tohoto rozhraní se metodá volá rekurzivně a vyhodnocují se tak postupně podkonstrukce. Jakmile je třeba vyhodnotit samotnou aplikaci funkce na argumenty, je zavolána odpovídající metoda (`performAdition(String op1, String op2)`,`performSubtraction(String op1, String op2)` atd.), která vrátí výsledek, jenž pak může být dále zpracován.

Třída UnrecognizedEntityException Tato třída představuje výjimku, která je vyhozena v případě, že interpret narazí na neznámou konstrukci či jinou entitu (například neznámou funkci). Veškerou funkcionalitu poskytuje rodičovská třída **Exception**, třída **UnrecognizedEntityException** při vytváření instance pouze zavolá nadřazené konstruktory, o zbytek se již postará zmíněná třída **Exception**.

5.2.2 Balíček *gui*

Třída TSFileFilter Třída **TSFileFilter** překrývá zděděné metody `accept(File file)` a `getDescription()` z třídy **FileFilter**.

Metoda `accept(File file)` vrátí `true`, pokud je daný soubor (reprezentovaný třídou **File** ze standardního API jazyka Java) požadovaným souborem s příponou „.ts“, případně „.TS“.

Metoda `getDescription()` vrací textový popis filtrovaných souborů (ten se objeví v dialogu s výběrem souboru).

Třída MainWindow Tato třída obsahuje prvky uživatelského rozhraní, které jsou reprezentovány třídami knihovny Swing (například textová pole – **JTextField**, tlačítka – **JButton** atp.) a integruje je do jediného okna, tvořeného třídou **JFrame** (taktéž z knihovny Swing), se kterým může uživatel po spuštění programu pracovat. Jednotlivé třídy a způsob jejich integrace zde nebudu popisovat, kód je generován (polo)automatický s pomocí návrháře rozhraní IDE NetBeans, v němž byla implementace realizována.

Nejzajímavějšími metodami v této třídě jsou `openSourceFileButtonActionPerformed(ActionEvent evt)` a `runButtonActionPerformed(ActionEvent evt)`, jejichž kód se vykoná, jakmile uživatel klikne na tlačítko „Open“ respektive „Run!“.

V prvním případě je využito prostředků třídy **JFileChooser**, která realizuje zobrazení dialogu s výběrem vstupního souboru. Díky námi definované třídě **TSFileFilter**, jejíž instance je předána instanci třídy **JFileChooser**, se v dialogu objeví pouze přípustné soubory (a složky). Po tom, co uživatel potvrdí svou volbu, se obsah zvoleného souboru, zdrojový kód TIL-Scriptu, načte do hlavního textového pole interpretu, kde jej uživatel může dále upravovat.

V případě metody `runButtonActionPerformed(ActionEvent evt)` je zkontrolováno, zda existuje vstup, nad kterým by interpret mohl být spuštěn, tj. zda se v hlavním textovém poli nachází nějaký text, případně, jestli uživatel nespecifikoval cestu

ke zdrojovému souboru „ručně“ v textovém poli k tomu určeném. Jsou-li obě tato pole prázdná, je na to uživatel upozorněn vyskakujícím oknem (realizovaným s pomocí třídy **JOptionPane**). Pokud je vše v pořádku, je vytvořena instance interpretu a na ní zavolána metoda `interpretITE(String source)` se zdrojovým kódem předaným v parametru `source`. Jakmile je práce interpretu skončena, je výsledek zapsán do textového pole pro výstup, kde si jej uživatel může prohlédnout.

5.3 Ukázky interpretace

Ukažme nyní na několika příkladech chování interpretu. Níže uvedeným ukázkovým programům v TIL-Scriptu, které ukazují chování funkce *If-then-else* a *If-then-else-fail*, odpovídají výstupy pod nimi. Každou ukázku popisuje krátký komentář.

Příklad 5.1
První dvě ukázky ilustrují chování funkce *If-then-else*. V ukázce č. 1 je podmínka vyhodnocena jako nepravdivá a je tedy vybrána a provedena druhá větev funkce, což vede k výsledku „30“.

V ukázce č. 2 je podmínka pravdivá, vybrána a provedena je tedy první větev, ta však *v*-konstruuje nevlastní *Kompozici*, neboť nulou dělit nelze, proto funkce selhává (nevrací žádnou hodnotu).

```
# ukazka c. 1
['If_then_else
    ['Or
        ['= '35 ['* '4 '8]]
        ['Not 'True]
    ]
    '['+ '17 '33]
    '['* '6 ['/ '20 '4]]
].
Result #1: 30.

# ukazka c. 2
['If_then_else
    ['And
        'True
        ['Implies
            'False
            ['= '14 ['* '2 '7]]
        ]
    ]
    '['/ '77 '0]
    ''61
```

```
].
Result #2: Construction failed!
```

V případě ukázky č. 3 je výsledkem *Provedení* druhého argumentu, tj. „75“, jelikož byla podmínka vyhodnocena jako pravdivá.

Podmínka v ukázce č 4. je nepravdivá, funkce *If-then-else-fail* tedy selhává a není vrácena žádná výstupní hodnota.

```
# ukazka c. 3
['If_then_else_fail
    ['Or
        ['= '6 '7]
        ['And 'True 'True]
    ]
    '['* '5 ['+ '6 '9]]
].
Result #3: 75

# ukazka c. 4
['If_then_else_fail
    ['Not
        ['Or
            ['= '6 ['* '2 '3]]
            ['Implies 'True 'False]
        ]
    ]
    '['+ '8 ['/ '144 '12]]
].
Result #4: Construction failed!
```

■

Tyto a další ukázky jsou součástí elektronických příloh. Ukázkové aplikace funkce *If-then-else* jsou umístěny v souboru *If-then-else.ts*, zatímco příklady funkce *If-then-else-fail* v souboru *If-then-else-fail.ts*.

5.4 Návrh řádné implementace

V průběhu implementace jsem narazil na problém při zachycování pythonovských výjimek prostředky jazyka Java, což může být důležité při ošetřování výjimečných stavů, které mohou nastat při parsování vstupu. Další nevýhodou je obtížná úprava stávajícího kódu parseru, který se k projektu připojuje v jar archívu. Vzhledem k výše popsané problematické integraci stávajícího parseru napsaného v Pythonu do jiných jazyků, bych doporučil přepsat jej do jazyka, ve kterém bude implementace realizována (předpokládá se budoucí implementace v jazyce C# nebo Java).

Pro usnadnění budoucí implementace řádného interpretu TIL-Scriptu je zde prezentován možný objektový návrh (formou třídního diagramu), který vychází ze zkušeností s implementací jednoduchého interpretu funkce *If-then-else*, a ukazuje možnou budoucí strukturu kódu interpretu.

Poznámka 5.2 Popis třídního diagramu předchází samotný diagram (na obrázku 3), ten je umístěn až na další straně z důvodu jeho větší velikosti.

5.4.1 Třídní diagram

Třída **TSParser** představuje parser TIL-Scriptu, pro úsporu paměti je dovoleno vytvořit pouze jednu instanci této třídy (návrhový vzor Singleton), na této instanci lze poté volat metodu `parse(String sourceCode)`[21], která vrátí uspořádanou dvojici **Pair<List<ASTNode>, Environment>**. Prvním prvkem této dvojice je seznam kořenových uzlů AST odpovídajících jednotlivým výrazům v TIL-Scriptu, které měl parser zpracovat. Druhým prvkem je prostředí, tj. objekt držící seznam uživatelských deklarací v TIL-Scriptu (proměnné, pojmenované funkce). Předpokládá se, že parser před ukončením parsování vstupu provede i typovou kontrolu (to však není striktně určeno, je zde prostor pro alternativy).

Třída **Interpreter** realizuje samotnou interpretaci – metodě `interpret(ASTNode root, Environment env)`, je důležité předat, kromě kořenového uzlu AST, i objekt třídy **Environment**, aby se případné výskyty (globálních) proměnných či pojmenovaných funkcí mohly správně vyhodnotit. Tato třída také využívá návrhového vzoru Singleton.

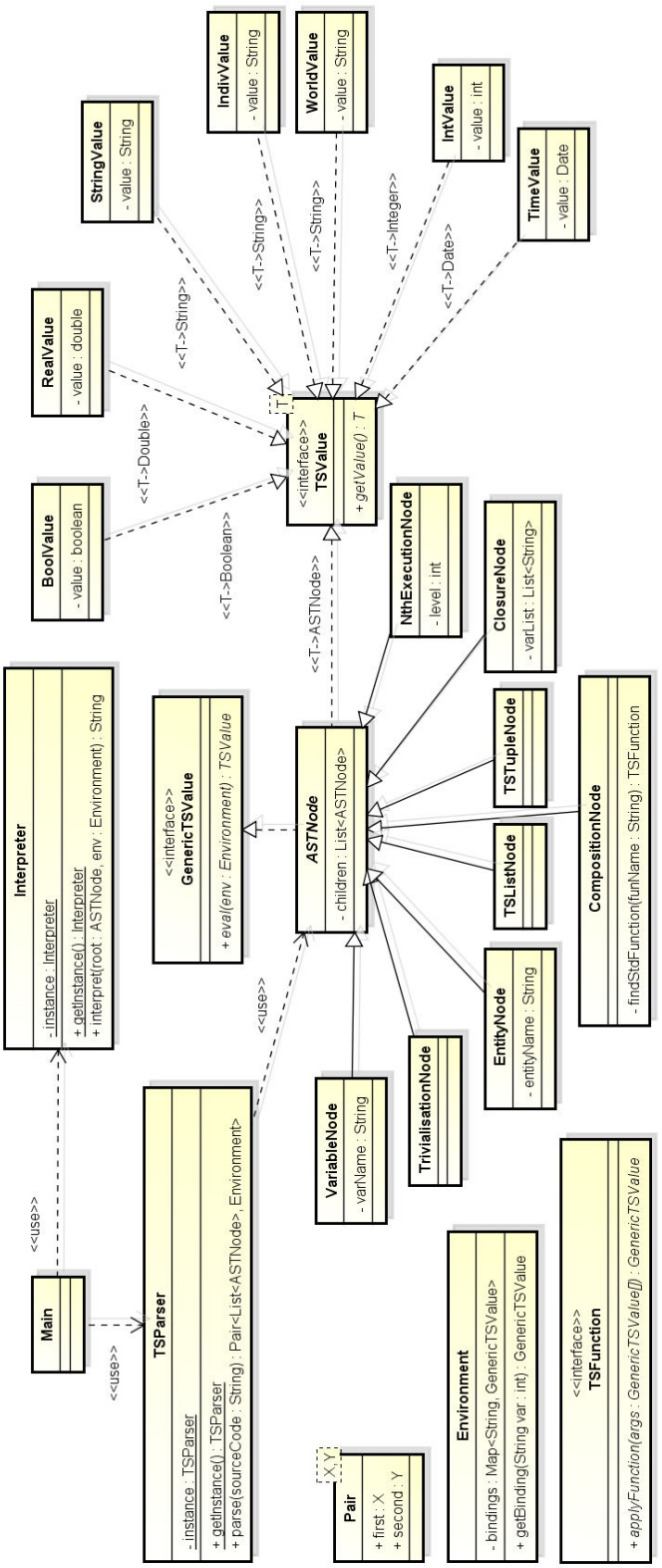
Třída **Main** představuje vstupní bod (metoda `main(String[] args)` však v diagramu není vyobrazena) celého interpretu a kloubí funkcionalitu obou výše popsaných tříd. Nejprve získá AST díky instanci třídy **TSParser** a poté je postupně nechá interpretovat s pomocí instance třídy **Interpreter**.

Abstraktní třída **ASTNode** reprezentuje uzel v AST. Součástí uzlu jsou vždy reference na jeho potomky (atribut `List<ASTNode> children`) jednotlivé třídy rozšiřující tuto třídu (**VariableNode**, **TrivialisationNode**, ...) představují typy konstrukcí, jak je známe z TIL, a přidávají funkcionalitu specifickou pro konkrétní typ konstrukce. Ve třídě **CompositionNode** nalezneme metodu `findStdFunction(String funName)`, která se stará o nalezení příslušné funkce *Kompozice* v knihovně standardních funkcí, která bude součástí jazyka TIL-Script (*If-then-else* budiž příkladem takovéto standardně definované funkce). Tato funkce je reprezentována rozhraním **TSFunction**, které umožní volat metodu `applyFunction(GenericTSValue[] args)` a aplikovat tak funkce na argumenty předané v parametru `args`.

Jelikož TIL pracuje i s objekty vyšších řádů než 1, je nutné s tímto faktem počítat i při návrhu interpretu TIL-Scriptu. Pro tyto účely je zde rozhraní **GenericTSValue**, které třída **ASTNode** implementuje. Zmíněné rozhraní předepisuje implementaci metody `eval(Environment env)`, která je použita v případě, že daná konstrukce ještě není výslednou hodnotou a je třeba ji provést. Kromě tohoto rozhraní však třída **ASTNode** implementuje i rozhraní **TSValue<ASTNode>** definující metodu `getValue()`,

[21] V textu uváděné zápisy hlaviček metod používají syntaxi jazyka Java, v diagramu je notace mírně odlišná.

jež je použita v případě, že daná konstrukce již má být výsledkem evaluace. Parametrizované rozhraní **TSValue<T>** (kde **T** je typem hodnoty) poté implementují objekty typu řádu 1, které známe z TIL, v TIL-Scriptu jsou to objekty typu Bool, Indiv, Time, World, String, atd. Každý takový objekt si drží vlastní hodnotu přístupnou po zavolání metody `getValue()`, která již byla zmíněna v souvislosti s třídou **ASTNode**.



Obrázek 3: Třídní diagram pro interpret jazyka TIL-Script

6 Závěr

6.1 Dosažené cíle

Jedním z hlavních cílů práce bylo popsat problematiku specifikace funkce *If-then-else*. Byla uvedena definice *If-then-else* ve smyslu logické spojky a logického operátoru v predikátové logice 1. řádu a vysvětlena jejich sémantika, následně bylo zjištěno, že takováto definice není zcela dostačující a nesplňuje nároky na chování této funkce v případě, kdy je některý z argumentů nedefinován. V následující sekci byla funkce *If-then-else* rozebrána z hlediska teorie programovacích jazyků. Po zavedení několika nezbytných pojmů, zejména redukční (evaluační) strategie a její striktní a lazy varianta, bylo na názorném příkladu programu v jazyce Haskell, jenž nám díky své jednoduché sémantice výborně posloužil pro demonstraci, ukázána nemožnost definování non-striktních funkcí na uživatelské úrovni v jazycích se striktní evaluací (tj. i funkce *If-then-else*). Práce se poté zabývala definicí *If-then-else* v TIL, kde je díky hyperintenzionalitě a parcialitě zajištěno požadované chování, které je navíc v souladu s principem kompozicionality. Dále kapitola pokračuje analýzou vět spojených s presupozicí, větami v minulém a budoucím čase, kde je mj. představeno obecné schéma analýzy těchto vět. Nakonec je uvedena ukázka komunikace agentů v multiagentním systému, na které je dobře vidět, že je TIL vhodným prostředkem takovéto komunikace, celá ukázka byla také převedena do jazyka TIL-Script.

Součástí práce je také shrnutí základů Transparentní intenzionální logiky a jejích základů. Práce také představuje nejnovější podobu jazyka TIL-Script, je zde podrobně popsána jeho syntax, která stojí na revidované verzi gramatiky, která je taktéž součástí této práce.

Kvůli ranému stádiu vývoje jazyka TIL-Script nebylo možné vytvořit modul, který by se integroval do již fungujícího interpretu jazyka TIL-Script, neboť ten v době psaní práce dosud neexistoval (před započetím práce byl k dispozici pouze parser). Pro demonstrační účely byl tedy vytvořen jednoduchý interpret funkce *If-then-else* v TIL-Scriptu, jehož implementace i uživatelské rozhraní bylo popsáno. Byl také vypracován objektový návrh budoucí implementace řádného interpretu, který je v práci také popsán.

6.2 Pohled do budoucna

Další vývoj projektu funkcionálního jazyka TIL-Script bude zcela určitě obnášet implementaci řádného interpretu, přičemž může být využito objektového návrhu uvedeného v této práci. Postupně by se měly možnosti tohoto jazyka rozšiřovat o následující věci:

- Implementace standardní knihovny běžně používaných funkcí, která bude součástí API jazyka TIL-Script.
- Návrh vhodné reprezentace znalostní báze a schopnost jejího využití v rámci jazyka TIL-Script.
- Vytvoření inferenčního stroje pro TIL-Script, který umožní vyvozování důsledků z explicitně daných faktů (uložených ve znalostní bázi).

7 Reference

[1] RACLAVSKÝ, Jiří. Pavel Tichý: Curriculum vitae. *Homepage of Transparent Intensional Logic* [online]. 2007 [cit. 2014-02-05]. Dostupné z: `http://til.phil.muni.cz/text/biography_tichy.php`

[2] DUŽÍ, Marie a Pavel MATERNA. *TIL jako procedurální logika: průvodce zvídavého čtenáře Transparentní intensionální logikou*. 1. vyd. Bratislava: Aleph, 2012, 412 s. Noema, 7. sv. ISBN 978-808-9491-087.

[3] TICHÝ, Pavel. *The Foundations of Frege's logic.* Berlin: Walter de Gruyter, 1988, xiii, 303 s. ISBN 3-11-011668-5.

[4] CIPRICH, Nikola. *TIL-Script – The Computational Variant of TIL*. Ostrava, 2009. Diplomová práce. Vysoká škola báňská – Technická univerzita Ostrava. Vedoucí práce doc. RNDr. Marie Duží, CSc.

[5] ISO/IEC 14977. *Information technology – Syntactic metalanguage – Extended BNF*. Ženeva: International Organization for Standardization, 1996. Dostupné z: `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`

[6] MANNA, Zohar. *Matematická teorie programů*. 1. vyd. Praha: SNTL, 1981. 467 s. Knižnice výpočetní techniky.

[7] Strict function. *Wikipedia, The Free Encyclopedia*. [cit. 2014-04-10]. Dostupné z: `http://en.wikipedia.org/w/index.php?title=Strict_function&oldid=603203513`.

[8] Haskell/Denotational semantics. *Wikibooks, The Free Textbook Project*. [cit 2014-04-10]. Dostupné z: `http://en.wikibooks.org/w/index.php?title=Haskell/Denotational_semantics&oldid=2622770`.

[9] Expression (computer science). *Wikipedia, The Free Encyclopedia*. [cit. 2014-04-10]. Dostupné z: `http://en.wikipedia.org/w/index.php?title=Expression_(computer_science)&oldid=598667897`.

[10] ŠKARVADA, Libor. *Úvod do funkcionálního programování: studijní text* [online]. Masarykova univerzita, 2002 [cit. 2014-04-10]. Dostupné z: `http://haluska.tym.sk/2.%20rocnik/fp/funkcionalne_programovanie-CZ.pdf`.

[11] ŠKARVADA, Libor. *Doprovodný text k přednáškám* [online]. Masarykova univerzita, 2010 [cit. 2014-04-10]. Dostupné z: `tomi.nomi.cz/tmp/skola/IB015/sl.pdf`

[12] DUŽÍ, Marie, Martina ČÍHALOVÁ a Marek MENŠÍK. Communication in a multiagent system: questions and answers, *SGEM 2013*, STEF92 Technology Ltd, Albena, 11-22. ISBN: 978-954-91818-9-0

[13] ČÍHALOVÁ, Martina, Marie DUŽÍ a Marek MENŠÍK. Communication in a Multi-Agent system Based on Transparent Intensional Logic. *MENDEL 2011*, Ed. Matoušek, R., 477-485. Brno: VUT.

[14] DUŽÍ, Marie. Tenses and truth-conditions: a plea for if-then-else. *The Logica Yearbook 2009*. Ed. Peliš, M. King's College London: College Publications, 2009, 63-80.

[15] *Java™ Platform, Standard Edition 7: API Specification* [online]. ©1993-2014 [cit. 2014-04-12]. Dostupné z: `http://docs.oracle.com/javase/7/docs/api/`

[16] Jython. *Wikipedia, The Free Encyclopedia*. [cit. 2014-04-12]. Dostupné z: `http://en.wikipedia.org/w/index.php?title=Jython&oldid=602075492`

A Příloha na CD

- Jednoduchý interpret funkce *If-then-else* pro TIL-Script.
- Soubory *If-then-else.ts* a *If-then-else-fail.ts* obsahující ukázkové aplikace funkce *If-then-else(-fail)* pro zpracování implementovaným interpretem.
- Soubor *mas_com_example.ts* soubor s ukázkou komunikace agentů převedenou do jazyka TIL-Script.